Match and Move, an Approach to Data Parallel Computing

Thomas J. Sheffler
October 1992
CMU-CS-92-203

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**A Dissertation
Submitted to the Department of Electrical and Computer Engineering
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy**

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Connection Machine time was provided by the Northeast Parallel Architectures Center, and by the Pittsburgh Supercomputing Center, grant number ESC910004P. Cray Y-MP time was provided by the Pittsburgh Supercomputing Center (Grant ASC890018P).

Keywords: Data-parallel, parallel algorithms, vectorized algorithms, massive parallelism, supercomputing

Abstract

The **match** operation, introduced by G. Sabot, is a parallel primitive describing a communication pattern. For two vectors of keys, called the "source" and "destination," a path is indicated for each element of the source to each destination element having an equal key. The result of the **match** operation is a data structure called a "mapping" encapsulating these paths. A **move** operation sends data from the sites of a source data vector through the paths of the mapping to a destination data vector. If the mapping is many-to-one, a combining function, such as **add**, **max** or **min**, specifies how collisions are resolved at the destination sites.

The **match** and **move** operations provide a unified set of parallel primitives that at once satisfy the conflicting demands of generality and efficiency. This dissertation illustrates the generality of the approach by showing many algorithms implemented with **match**, focusing on algorithms for graphs, sparse matrices, and binary tree structures. These types of problems are characterized as involving dynamic structural change, and are typically difficult to implement on parallel architectures. By using the framework allowed by the **match** operation, the algorithms are straightforward and simple to understand.

Because **match** and **move** may be implemented efficiently on many different parallel and vector architectures, algorithms expressed in this manner achieve architecture independence. In addition, because there are many implementation strategies for the operators available for a given machine, they provide a level of implementation independence. This level of abstraction is supported without sacrificing absolute performance however. We support this claim by presenting data collected from implementations on the CRAY Y-MP and the Connection Machine CM-2. Our programs are competitive with more traditional methods of implementation, while remaining high-level and architecture-independent.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

This work owes much of its inspiration to Gary Sabot, whose Paralation Model of computing captured my imagination. It was while attempting to implement parallel programs that I came to the conclusion that his was the right way to think about parallel programming. I hope that my investigation of **match** and **move** serve to amplify the claims he made in his dissertation. Guy Blelloch has had an enormous impact on my thinking as well. Much of what I have explored has built on his ideas. As a friendly critic, his comments and suggestions were invaluable as this work progressed. I also owe a great debt to Allan Fisher and Rob Rutenbar, who were instrumental in helping to focus this work — probably more than they realize.

My sincere appreciation goes to my advisor, Randy Bryant, for the many ways he helped me through the last few years. As well as being an excellent technical advisor who was always willing to help me sort through my muddled ideas, it was through his example that I learned the most about excellence in research. I am grateful to him for his guidance and enthusiasm throughout the course of my stay at CMU.

Chapter 1

Introduction

In the continuing attempt to develop computers that offer higher performance, advanced supercomputer architectures are evolving through the addition of increasing numbers of processors and vector units. Because of the diversity of models these machines present, programmers traditionally have targeted their programs specifically for each type of parallel or vector computer. In addition to complicating the task of programmers, this approach severely limits the portability of applications developed for any particular machine. While some standard programming tools are emerging, they are not universally accepted. High-level constructs must be developed that simplify the views of these architectures. However, it is equally important that abstract software layers do not sacrifice performance, or they will be avoided.

Some progress has been made for programming tools aimed at a limited range of problems; these are generally those with a regular structure. Unfortunately, no such consensus exists for problems of irregular and dynamic structure. Very little programming support targets algorithms on graphs, sparse matrices and trees: structures that change during the course of an algorithm. As a result, they are handled differently for each machine. While a great amount of computer science research concerns algorithms for these data structures, there has been a dearth of activity in their practical use. Part of the problem may be the very fact that their implementation using existing programming tools is so difficult that they are simply not attractive problems.

The research presented here proposes a unified framework for expressing algorithms on problems of dynamic structure that offers high performance on a variety of parallel architectures. This chapter first describes why portable parallel programming is a difficult problem and then introduces **match** and **move** as operators of sufficient generality to be applied to many different architectures for solving many types of problems. It then compares this approach to current developments in parallel programming models, architectures, algorithms and languages. This discussion sets the stage for the outline of the rest of the dissertation.

1.1 Overview

This dissertation proposes a way to write high-performance portable parallel programs. Writing parallel programs that are portable is difficult because parallel computers vary greatly in their capabilities and features. While all parallel computers share the characteristics that they comprise some number of processors and an interconnection network for communication, an enormous number of implementation variants of even these basic features complicates the problem. One approach to writing a portable parallel program uses only those features common to many machines. However, this approach suffers from the drawback that it uses only the most general capabilities of each machine causing special features available on a particular machine to go to waste. For example, some machines permit multiple processors to read from the same location, but because many machines do not, a truly portable program could not be designed using such a feature.

Typically, programmers sacrifice portability for high performance. They write their programs at a very low level, making use of specialized features of a target machine. In addition to making a program inherently non-portable, this approach to programming is laborious and error prone. Having to consider low-level details along with high-level issues, a programmer can quickly reduce an elegant algorithm to an ugly and confusing program that is difficult to debug.

Raising the level of abstraction at which a programmer views a parallel machine enhances portability. However, while heightened abstraction simplifies programming, it may also adversely affect performance. Parallel machines are designed and purchased to run applications at very high speed, and even a beautiful abstract software layer will not find acceptance if it sacrifices high performance.

This dissertation proposes a way to write portable parallel programs using two abstract operators called **match** and **move**, as proposed by Gary Sabot in his Paralation model of computing [Sab88b]. Because parallel machines differ mainly in the design of their interconnection networks and processor configurations, the main hindrance to making parallel programs portable is describing data movement operations in a very general manner. Once data is moved to the place where it is needed for computation, most parallel machines provide similar assortments of arithmetic and logical operations. **match** and **move** provide the means to describe data movement in an architecture-independent manner.

1.1.1 match and move

The **match** operator describes channels of communication between processing sites, and **move** sends data through the paths described by **match**. Rather than using processor numbers or memory addresses, the **match** operator generalizes the description of an address to the simple notion of a key. In this manner, data are identified in a way that is independent of the number of processors or

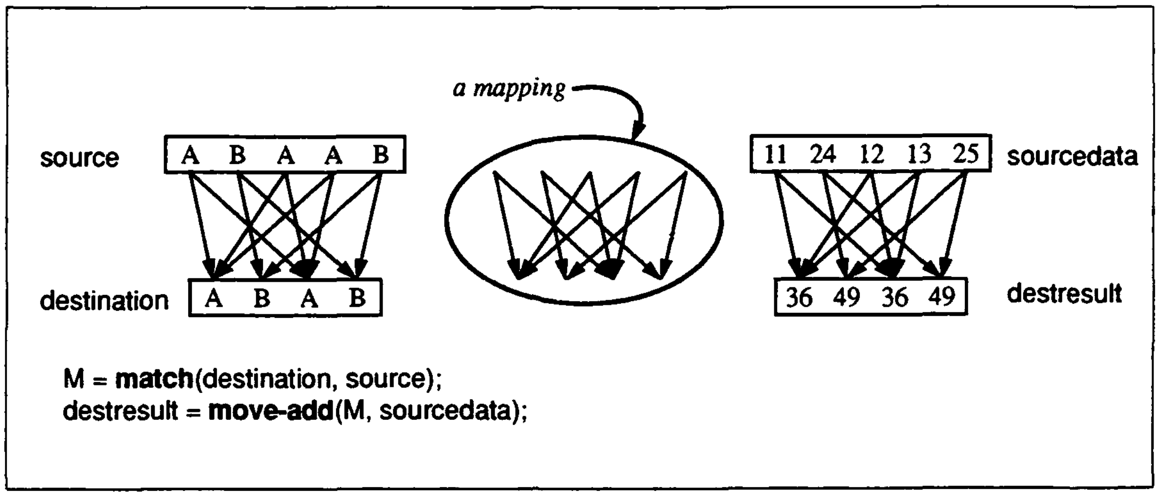


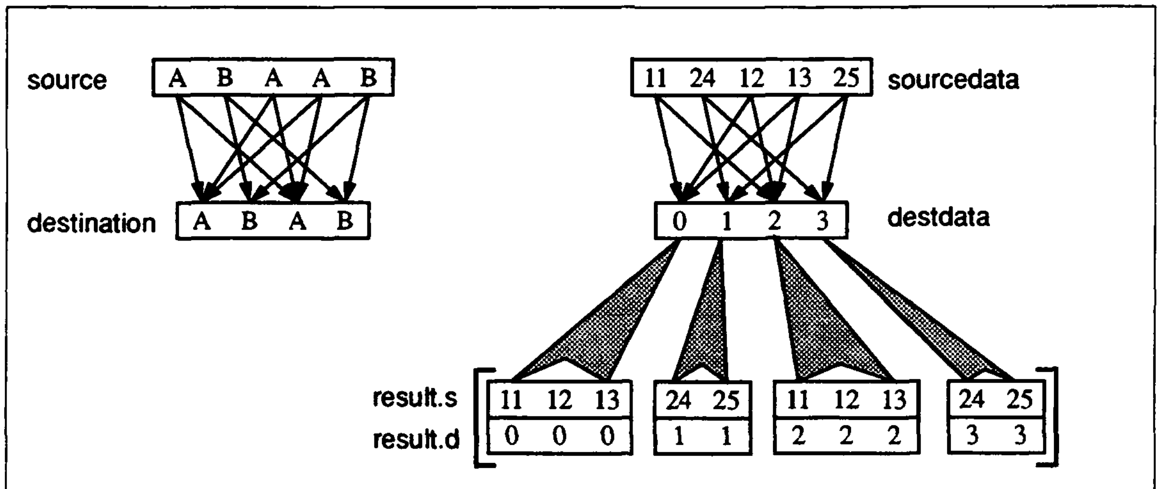
Figure 1.1: The **match** operation. The **match** operation constructs a mapping from a source vector to a destination vector. A **move** operation can use this mapping to send data from a source vector to a destination vector with a combining function.

their interconnections. Two vectors called the "source" and "destination" define a communication pattern. For each element in the source, a path is indicated to each destination element having an equal key. The result returned by the **match** operation, called a "mapping," is used with a **move** operation to send data through these paths. Variants of the **move** operator generalize the ways in which data are rearranged in parallel programs to include permutations, combinations, and cross-products.

The **match** operation of Figure 1.1 shows two vectors used to create a mapping. Arrows represent the communication paths of the mapping that are set up between equal keys. Because duplicate keys may exist in both the source and destination vectors, mappings may describe complex patterns. When multiple source sites map to the same destination, variants of **move**, such as **move-add**, **move-max** and **move-min**, specify combining functions for data values. In Figure 1.1 the mapping is applied to a vector sourcedata with the **move-add** operation. The result vector contains the sums of values from source sites sharing the same key.

Data values from matching source and destination sites may also be brought together in a new vector with the **move-join** operator. An example **move-join** operation is shown in Figure 1.2. **move-join** provides a combining function that arranges the values colliding at a destination site into adjacent sites in a new vector. For simplicity, the **move-join** operation is defined to return a vector of tuples. A tuple is an ordered set of values whose member fields are identified by position. The two fields of the result vector relate each site to a pair of values from the data vectors sourcedata and destdata given as arguments to **move-join**.

The family of **match** and **move** operators generalize many of the ways in which data are rearranged in parallel programs. **match** extends the notion of an address from that of a constant



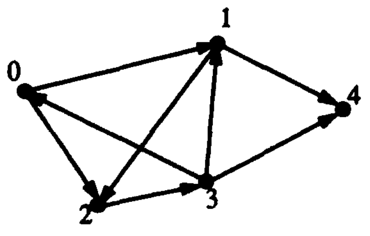
```
M = match(destination, source);
result.<s,d> = move-join(M, sourcedata, destdata);
```

Figure 1.2: The **move-join** operation. The **move-join** operation provides a combining function that arranges the values colliding at a destination site into adjacent sites in a new vector. The two fields of the result vector relate each site to a pair of sites from the source and destination vectors.

site identifier to the more general mechanism of variable keys. In addition to the one-to-many and many-to-one patterns typical for parallel read and write operations, keys may describe many-to-many patterns as well. Encapsulation of patterns in mapp' ¸s also allows them to be re-used, often amortizing a large set-up cost over many instances of its use. The **move** operators generalize the methods by which data values are permuted or combined. The simple combining **move** operators perform arithmetic functions, while the **move-join** performs a structural combination function.

1.1.2 Algorithms

In this approach, a data structure is represented by an unordered collection of tuples. The collection is stored as a vector, and values belonging to the same tuple are at the same processing "site." For example, a simple directed graph is represented as a collection of vertices and edges that each have a tail and head field. The following example shows a graph with vertices labeled from 0, but general keys could be used as well.



verts.#	0	1	2	3	4			
edges.tail	0	2	3	0	1	1	3	3
edges.head	1	3	0	2	4	2	1	4

Mappings express relationships between the unordered elements of the data structure. In contrast to other approaches where programmers must often arrange data into some predefined order, mappings simplify data structure definition. For instance, in a typical parallel connected components algorithm, an important step requires each vertex to find the minimum numbered vertex of its neighbors. Using **match** and **move**, this operation is implemented in two steps.

```
M = match(verts.#, edges.head);
v.minneighbor = move-min(M, edges.tail);
```

The **match** statement creates a mapping relating each edge to the vertices for which it is a head. This mapping is many-to-one, since vertices may have multiple incoming edges. The **move** statement uses the mapping, M, to send the tail value of each edge to those vertices for which it is an incoming edge. When multiple values are sent to the same vertex, the minimum value is chosen.

Programs written using **match** and **move** use mappings to express relationships between the elements of data structures. In this simple example, the mapping, M, grouped edges together that have the same head. Mappings make such relationships explicit, rather than a side effect of some other computation such as sorting the edges by their head value. This simple algorithm will be explored in other programming models later where the same result will be computed but in a less direct manner. In addition to obscuring the desired relationships, these other approaches will be shown to inherently non-portable.

This dissertation illustrates the power of **match** and **move** with this simple data representation through a large number of example algorithms. Algorithms on sets, sparse vectors, graphs, binary trees and sparse matrices are handled easily in this unified framework. In addition to simplifying their description, algorithms written using **match** and **move** achieve the same algorithmic efficiency measures of many of the best known parallel algorithms.

1.1.3 Implementation

The portability of the approach to parallel programming presented here relies on the efficient implementation of **match** and **move** on many types of parallel architectures. A large part of this dissertation discusses in detail the implementation of the **match** and **move** operators in various theoretical models of computation and also on two different parallel machines, the Connection Machine CM-2 and the CRAY Y-MP. The two machines chosen represent two very different architectures, a massively parallel computer and a vector supercomputer.

Two main approaches to implementing the general purpose **match** operator are identified. One is based on sorting and the other on hashing. While hashing has a long history in the folklore of parallel computing, its use has often been regarded with skepticism because of its probabilistic nature. This dissertation compares the two approaches and shows that hashing is a viable alternative to sorting for many problems. This claim is supported by data collected from the implementations on both machines.

The implementation of the full suite of **match** and **move** operators required the development of new algorithms, some of which are of general purpose use. This document explains these algorithms and carefully studies their algorithmic complexity and actual performance figures. This emphasis on implementation strives to separate this research from being a purely theoretical exercise and proves that it is a viable approach for many real machines.

1.1.4 Focus of this Dissertation

This dissertation presents **match** and **move** as a portable approach to data parallel programming for problems of dynamic structure, offering enhanced architecture independence over many other parallel primitives. The ease with which algorithms may be designed in this model is illustrated by a large number of example algorithms built on **match**. The brevity of the algorithms described in this manner illuminates the power of the **match** operation.

This dissertation extracts the **match** and **move** primitives from Sabot's Paralation model. Paralation Lisp is an expressive language for writing parallel algorithms, but it has not been widely used because a full implementation exists only as a simulator. This research intends to show that even without a full language implementation, **match** and **move** are useful for many algorithms and can offer good performance for real problems on real machines.

A central point of the Paralation model not addressed is the management of nested parallelism. This issue has implications related to linguistic support for parallel programming as well as the exploitation of locality. The model provided herein is essentially "flat." There is nothing that precludes incorporating this work back into the full Paralation model, but focusing only on **match** and **move** emphasizes their utility outside of a full programming language.

This work also differs from Sabot's in the presentation of the **move-join** operator. In his Paralation model, Sabot provided a similar operator called **collect**. While the **collect** operator of the Paralation model provided the same functionality, its significance was not fully realized there. In his examples, Sabot used **collect** to rearrange the elements of a vector into segments. However, viewing it as a generalized cross-product generator more fully exploits its strength. The recognition of this capability allows it to be used in many algorithms in many different forms.

A prime concern is the actual performance that programs written using **match** and **move** deliver on real machines. In short, the thesis can be summarized in the following statement:

> **match** and **move** can be used to write architecture-independent parallel programs for problems of dynamic structure that achieve performance rivalling that of more traditional means.

Implementation strategies for the **match** and **move** operations are discussed for different target architectures and upper bounds are placed on the algorithmic complexity of the implementation of the basic set of operators. However, this research also develops a number of important optimizations that guarantee theoretical efficiency for many special circumstances. Aggressive implementations of **match** and **move** based on both sorting and hashing for each of the Connection Machine CM-2 and the CRAY Y-MP show that the operators are both architecture and implementation independent. The validity of this approach is proved by comparing the performance of algorithms using **match** and **move** to some industrial-strength numerical subroutines where the results presented show that this high-level approach offers comparable speed. While many proposed parallel programming models simply provide asymptotic complexity measures, the performance figures provided give compelling evidence that this approach offers high performance for real problems.

1.1.5 A Confluence of Ideas

This dissertation unifies various streams of thought running through the parallel computing community. Necessarily, it considers parallel programming models, architectures, algorithms and languages. The following sections observe trends with respect to each of these separate issues and then relate them to the parallel programming model based on **match** and **move**. The approach to parallel computing presented here is well suited to the many types of parallel supercomputers being developed, and shares much in common with an emerging consensus on the ways in which they should be programmed.

1.2 Parallel Machine Models

Currently, most parallel algorithms are designed for the theoretical PRAM (Parallel Random Access Machine) models. The PRAM models make explicit the number of processors and allow every processor to perform an arithmetic operation, or to read or write to a global memory in a single time step. This family of models is differentiated by the manner in which memory access collisions are handled. The basic EREW (Exclusive-Read, Exclusive-Write) PRAM model insists that no two processors access the same memory location at the same time, while CR (Concurrent-Read) and CW (Concurrent-Write) variants allow multiple processors to read from or write to the same location. Further variants differentiate the way in which concurrent write operations to the same memory location are resolved. The CRCW-ARB model stores an arbitrary one of the values written to the same location while the CRCW-PLUS model combines values. While the theoretical PRAM models capture some essential properties of parallel machines, they do not provide intellectual leverage to ease the task of writing portable parallel programs. Furthermore, these theoretical models do not reflect the performance of many real machines.

Because programs written for these models assume a specific number of processors and a memory access collision policy, they are generally non-portable. Even though theoretical simulation methods exist that allow CRCW algorithms to run on an EREW machine for instance, these results are of little practical use to programmers. The asymptotic measures of the complexities of these simulations hide important constant factors that must be considered for real applications.

The Parallel Vector models proposed by Guy Blelloch are another family of parallel models [Ble90]. Instead of an explicit number of processors, operations on varying sized vectors have their work evenly distributed over whatever processors are available. In essence, each element of a vector is assigned its own "virtual processor." In contrast to the PRAM family, these models are also synchronous, where parallelism arises from operations on entire vectors that may be performed in parallel. This is a style of data-parallel programming where programs have a single thread of control and where parallelism is exploited by operations across large sets of data. The advantage of the data-parallel style over others is that programs naturally scale to larger data sets.

Programs written in this manner are less tightly coupled to a particular machine configuration or topology. However, because the Parallel Vector models also have EREW and CRCW variants, programs written for these models are tied to a particular memory collision policy. Thus, programs written for these models either make use of advanced features of a particular machine and become non-portable, or use only the most general capabilities and possibly sacrifice performance.

Rather than proposing a new theoretical model, this dissertation demonstrates a *practical* parallel programming model. It is built on top of the Parallel Vector model using the notions of vectors and virtual processors. In contrast with the Parallel Vector models, this practical approach combines all of the memory access methods under one general mechanism, namely the **match** and **move** operators as proposed by Gary Sabot [Sab88b]. Algorithmic complexity is measured using the metrics of the EREW Parallel Vector model, but actual implementations of **match** and **move** may transparently make use of advanced architectural features on a given machine, guaranteeing high performance for real applications.

This approach simplifies the programming problem and ensures architecture independence. The following section emphasizes this point by revisiting the simple graph algorithm shown earlier. While the approach using **match** and **move** is very direct, algorithms in the other models are less direct and inherently non-portable.

1.2.1 Algorithms

A simple parallel algorithm was presented earlier that required each vertex in a graph to find the neighbor with the minimum index. Recall that using **match** and **move**, this operation was implemented in two steps.

```
M = match(verts.#, edges.head);
v.minneighbor = move-min(M, edges.tail);
```

There are many ways to implement this operation in some of the parallel models mentioned above. With the combining write of the CRCW-PLUS PRAM model, a single step suffices to find the minimum tail for each group of edges with the same head. If the combining write is not supported, sorting might be used to find the minimum of each group of edges instead. Both of these approaches dictate a detailed implementation strategy however, rendering the resulting program non-portable. **match** and **move**, on the other hand, describe the desired result of the computation in a more general manner and allow the implementation to choose an efficient method of execution.

To make this point more concrete, this same algorithm is examined as it would be expressed in some of the other models. In the strict EREW PRAM model, a typical approach would use sorting to find the minimum valued tail for each group of edges with the same head key. The following sequence of steps describes such an algorithm.

1. Assign each vertex and edge to a separate processor.
2. Sort the edges by head value, and then by tail as a secondary key.
3. In the sorted order, divide the edges into groups wherever a new head value begins.
4. Because of the sorted order, the first member of each group is the minimum tail for a head. Send these values to the processor for the head specified.

The CRCW-PLUS variant of the PRAM models supports combining write operations. The PLUS operation is generalized to a number of binary associative operators including **min**. Writing the algorithm in this model is very simple, but its implementation is non-portable because few machines directly support such an operation. The following two steps would suffice.

1. Assign each vertex and edge to a separate processor.
2. In parallel, write the tail value to the processor identified by the head value using a **min** policy to resolve write collisions.

Finally, in the Parallel Vector model using segmented scans, a different approach might be used. A scan operation computes for each result element, the sum of all preceding elements with respect to a binary associative operator. Segmented vectors are simple vectors that are divided into contiguous sections called "segments." A segmented scan is a variant on a simple scan that begins a new running sum at each segment boundary. Thus, a segmented **min-scan** can be used to find the minimum value of each group of edges when the edges have been arranged into segments. The following steps describe an algorithm using segmented scans.

1. Sort the edges by head value.
2. Divide the edges into segments so that a new segment begins where the head value of an edge differs from the previous one.
3. Use a segmented **min-scan** function to find the minimum of each group of edges having the same head.

The theoretical models used for the examples above are useful for complexity analysis, but do little to aid the programmer. For this simple example it is not too difficult to wade through the pseudocode, but for more complicated algorithms it becomes increasingly difficult. Furthermore, these algorithms are very processor-specific and require a detailed understanding of the capabilities offered by a target machine. While a concurrent write with a **min** combining policy may be simulated on an EREW PRAM with $t = O(\lg n)$ slowdown, this fact does not provide much information about the real performance of such an algorithm to the programmer. Requiring a programmer to rethink these basic simulations at every turn increases the programming burden and jeopardizes the program's reliability. While **match** and **move** may in fact be implemented using means very similar to the algorithms presented above, there are also alternative strategies, based on hashing for example, that may not be immediately apparent. Writing a program using **match** and **move** allows a programmer to avoid many considerations of the detailed implementation strategy while focusing on high-level algorithm design.

1.3 Advanced Architectures

As computers evolve to offer faster clock speeds, greater throughput and larger storage capacity, advanced supercomputer applications are continually developed that require even greater speed. Rather than satisfying a desire for computing power, the availability of greater processing speed has made possible inquiry into new research areas that simply raise new questions. These, in turn, require even more computing power to be effectively explored. As each new generation of computer appears, applications emerge that strain its capabilities and spur the development of the next generation. Indeed, some of the most compute-intensive applications driving workstation development are the electronic CAD programs that verify and test designs of future computers. Current "Grand Challenge" problems are another example that are expected to drive the development of Teraflop supercomputers by the end of this decade.

In order to reach the goal of increased computing power, computer architectures are becoming more complex. Clock speeds will not continue to increase by orders of magnitude, and it is clear that further acceleration will be reached only through architectural advances. Currently, supercomputer architectures are evolving through the addition of processors and vector units. While massively parallel SIMD (Single-Instruction Multiple-Data) architectures are being explored in computers such as the Connection Machine CM-2, MasPar MP-1, and DAP, other manufacturers have been developing vector multiprocessors with increasing numbers of vector units and processors. These machines include the CRAY Y-MP, IBM 3090 VF and Alliant FX/80. More recently, Thinking Machines introduced the CM-5, a MIMD (Multiple-Instruction Multiple-Data) architecture that associates four vector units with each processor, and both Cray Research and Fujitsu have announced massively-parallel projects that will incorporate large numbers of processors and vector units [Bel92].

These architectures are becoming increasingly complex and more difficult to program. With a multitude of processors, vector units, a communications network and hierarchies of memory the programmer is met with a bewildering series of choices to make. It is increasingly important to adopt high-level parallel primitives that simplify the view of these architectures.

This dissertation proposes raising the level of abstraction of programs written for these machines by adopting the operators **match** and **move** as basic parallel building blocks. Because the common model that these machines support is one based on vectors, the Parallel Vector models are appropriate. However, the use of **match** and **move** to hide the many variants of communications primitives guarantees architectural independence. This approach is proven viable through aggressive implementations of the operators on both a vector supercomputer, the CRAY Y-MP and a massively parallel SIMD architecture, the Connection Machine CM-2. These results show that this high-level approach to supercomputing does not sacrifice absolute performance.

1.4 Programming for Problems of Irregular and Dynamic Structure

The majority of problems ported to parallel computers have been those of "regular" structure that are easily mapped to a particular arrangement of processors in a communication network. These problem categories tend to produce a static mapping of datum to processor, or use an unchanging communication pattern between data elements. Examples of such problems include a wide variety of algorithms expressed as dense matrix manipulations, cellular automata simulations [BE88] for image processing or fluid flow analysis, and neural network evaluation [Nor88].

A less widely explored class of problems includes those that have irregular or dynamically changing structure. These include algorithms on graphs, sparse matrices and trees. A common technique for many of these problems uses a pre-processing step to turn the irregular communication pattern of the original problem into a statically scheduled one [Dah90]. The solution of sparse linear systems is one such example.

On the CM-2, one approach [LMN$^+$] uses Parallel Nested Dissection [PR85] to construct a separator tree for the graph representing the system. The separator tree has at its leaves independent systems each involving a small subset of the original variables. Each of these smaller systems is mapped to the Connection Machine for factorization through a systolic algorithm. The results of each of the smaller systems are then combined at each of the higher levels until the solution to the entire system has been found. By using the front-end computer to find a separator tree and perform the mapping function, this algorithm made no use of the CM-2 to manage the structural changes occurring during factorization.

Other approaches for the CM-2 [Kra90] and a more general MIMD computer model [Bet86] completely solve the system symbolically on a serial computer. This pre-processing step produces an expression DAG (directed acyclic graph) that completely describes the computations to be performed in solving the system. A final step then maps the operators of the DAG onto the processors of the parallel computer. Good performance for the evaluation phase is obtained by requiring the mapping to use only simple communication. Once again, these implementations do not make use of the parallel facilities of the computer for handling the structural change of the underlying system.

Graph algorithms are another family of problems that rarely exhibit static structure; the graphs they operate on have a constantly changing shape. In these problems, which may be characterized as those of Dynamic Structure, the fundamental computations performed involve changes to the structure of a graph. When expressing these algorithms in a data parallel programming model, the mapping of graph element to processing site is no longer static and conventional programming techniques are not sufficient. However, the parallelism available in many graph algorithms suggests the potential for porting them to parallel machines.

Unfortunately, parallel programming support for these types of applications is limited. As will be demonstrated in this dissertation, the **match** and **move** primitives are an appropriate set of operations specifically oriented for programming applications for problems of irregular and dynamic structure. A large number of example algorithms support this claim by showing that **match** and **move** treat algorithms on graphs, trees and sparse matrices in a single unified framework.

1.5 Portable Parallel Programming

A desire for ease of programming accompanies the desire for faster and larger computers that can tackle increasingly larger and more complex applications. This seemingly contradictory wish arises because it is important that large applications remain intellectually manageable. Levels of abstraction that hide low-level details must be imposed so that programmers can grasp the important aspects of enormous application programs. However, software layers added solely to ease programming can adversely affect the performance of supercomputer applications, encouraging programmers to bypass them to obtain greater speed. Abstraction is acceptable only when it is not traded off against high performance.

Knowing that the next generation of computer will offer higher processing speed and will probably be available in just a few years, programmers also wish for programs that are easily adapted to new hardware. In the best of all possible worlds, programs could transparently make use of greater numbers of faster processors and larger memories without programmer effort. In fact, the cost in man-years that it takes to design and verify large supercomputer applications makes it important that code can be reused on advanced architectures. The next sections discuss some of the approaches currently being explored.

1.5.1 Portable Parallel Languages

Over the last few years, a debate that centered on the relative virtues of SIMD and MIMD computer architectures has persisted. Those who favored SIMD architectures argued that they were more easily scalable and simpler to program. Those who favored MIMD architectures argued for their flexibility and pointed out the inefficiencies of executing even simple branch statements across large datasets on SIMD computers.

A new view is emerging that proposes a Data-Parallel or Single-Program Multiple-Data (SPMD) programming model on either SIMD or MIMD computers. It presents a programming model that has a single thread of control whose operations scale to increasingly larger problem sizes. This synchronous model of execution means that there is only one locus of control, and that race conditions, deadlock, and nondeterminism are impossible. For programmers, this model is much simpler than other models involving concurrent processes and tasks. The issue of an implementation vehicle is separate, as it should be.

Current large-scale language development efforts targeting the SPMD model are those exploring dialects of FORTRAN 90 (FORTRAN-D [HKT92], CM-FORTRAN [Thi89], MPP-FORTRAN [PMM92]), the array syntax subgroup of the Numerical C Extensions Group (NCEG) working on standardizing C* [TPH92], and the NESL project at CMU [Ble92]. Each of these projects is proposing a programming model sufficient to capture data parallelism, and general enough to exploit this parallelism on many different architectures. Thinking Machines, for example, provides a single FORTRAN-90 compiler system that can operate on the CM-2, CM-5 or a single SPARC workstation. Each of these represents an implementation target whose run-time system is drastically different. However, the model presented to the user of FORTRAN-90 is the same across all three.

The SPMD approach to portable parallel programming proposes a common model across a wide variety of computer architectures. Since the compiler of these languages is required to perform data decomposition and task scheduling, the job of the programmer is made much easier. In addition, programs written in this style adapt to larger problem sizes and greater numbers of processors. This research does not propose a new language, but because **match** and **move** are data parallel operators, they could be easily integrated into any of the languages mentioned above.

1.5.2 Application Oriented Libraries

In January 1992, a panel discussion on the future of parallel programming language standards suggested multiple programming paradigms for parallel computers [Pan92]. While it seems that a common parallel language would benefit most users, Cherri Pancake argued that "... the move to standardization is self defeating. Most manufacturers have a standard language but also provide special features." Because programmers must exploit the special features provided by each vendor to get good performance, their programs are inherently non-portable. The move to FORTRAN-90 is encouraging, but it is unlikely that one universal language will satisfy all users and vendors. In that same discussion, Marc Snir suggested the adoption of application oriented libraries and program generators. One of the central problems however, is the delineation of the important operations for each application area.

In determining a unified approach across a wide variety of architectures, one of the most successful efforts has been the definition of the Basic Linear Algebra Subroutines (BLAS) for FORTRAN[LHKK79]. These subroutines represent the most commonly performed operations involving vectors and dense matrices in many numerical supercomputing applications. Their development is especially notable because it was driven by the *users* of supercomputers involved in numerical computations. As opposed to more abstract programming models proposed by compiler writers and computer scientists, the BLAS were developed to fill a current need. As a group, they represent an application-specific language for linear algebra on dense matrices.

The BLAS have been supported directly by many supercomputer vendors, often in the form of highly-optimized assembly code. This effort represents a high-performance programming model for dense matrix applications that is portable across a wide range of parallel computers. Because the subroutines are called from a single FORTRAN program but often result in parallel activity, they present an SPMD programming model.

The BLAS were defined in levels. Level 1 comprises operations on vectors ($O(n)$ complexity), level 2 are operations on vectors and matrices ($O(n^2)$ complexity), and level 3 are operations on matrices or entire linear systems ($O(n^3)$ complexity). The adoption of the BLAS themselves lead to new research fields: block methods for linear system solution followed directly from the availability of the Level-3 BLAS on a wide variety of architectures [ADD89]. It is conceivable that future application oriented libraries could spawn new ideas in other application areas.

This research suggests that **match** and **move** could be effectively used as a highly optimized library of functions called from other languages. Because implementations of **match** and **move** have many tuning parameters, highly optimized libraries of the operators might best be supported by computer vendors themselves. Just as the BLAS are an application oriented library for matrix applications, **match** and **move** may be used effectively as an application oriented library for applications on irregular and dynamic data structures.

1.5.3 Parallel Primitives

A large number of parallel primitives have been proposed for both real and theoretical machines. Parallel primitives are operations on aggregates of data values that can be performed by parallel algorithms. While some parallel primitives have been proposed for asynchronous PRAM models, this section focuses on their synchronous counterparts in a Data-Parallel framework. Programming models using parallel primitives need not be embedded in a language; frequently they are effectively used simply as a library of functions.

The Parallel Vector model of computing proposed by Guy Blelloch unifies a large number of parallel primitives in a synchronous (SPMD) framework. It has been successfully applied to both massively parallel SIMD machines and vector multiprocessors [Ble90, CBZ90]. The basic data structure of this model is the "collection," represented as a vector, with nested collections (i.e., collections of collections) represented as segmented vectors. Primitive communication operations include simple permutations through **send** and **get**, with data combining and reduction through **scan** operations.

More advanced parallel primitives of the Parallel Vector models include the combining-send and the multiprefix operation. The combining-send is a synchronous version of a concurrent write operation in which all values sent to the same location are combined by a binary associative operator such as **add**, **max** or **min**. The multiprefix operation is a synchronous version of the fetch-and-op operation of the NYU Ultracomputer[Ran87]. It can also be viewed as a keyed scan operation where groups are identified by key value rather than a segment descriptor. In fact, the multiprefix operation provides the functionality of the scan, segmented scan, and combining-send operations [Coh88].

Because the parallel primitives listed above may be emulated on many different machines, they can be used to write architecture-independent parallel programs. However, some of them require special-purpose hardware. For instance, the combining-send is supported by a data network on the CM-2, but it is not directly supported on the CM-5. The **match** and **move** operators provide many of the functions of these operators while they allow additional architecture independence.

Comparing match and move to Other Parallel Primitives

The functionality provided by **match** and **move** is similar to that of a variety of parallel primitives existing on real and theoretical machines. The common theme among the parallel primitives to be discussed is the association of each member of a collection of data values with one and only one group. For the **match** operation, groups are identified by keys. For the other primitives discussed, groups are identified by reference to a common variable, or by their arrangement in a vector.

On the Connection Machine, a combining-**send** operation identifies groups by parallel reference to a processing site. When using the **send-add** operation, for example, the final value written in the site is the sum of all values sent to it [Hil85]. The summing operation performed also generalizes to a number of other associative functions such as **max** or **min**.

The NYU Ultracomputer provides the **fetch-and-op** primitive [EGK+85, GLR81]. Groups are identified by a reference to a parallel variable. When executing a **fetch-and-op** on a parallel variable, the final value of the variable is the reduction (sum) of all the values sent to it, while each of the referencing sites receives one of the partial sums computed during the reduction. The order in which the partial sums is computed is unspecified. While this operation was designed as a synchronization primitive for an asynchronous programming model, it is of separate interest as a combining primitive in a data-parallel framework.

The multiprefix operator of the Fluent abstract machine subsumes the functionality of both the **fetch-and-op** and **scan** primitives [RBJ88]. It is also related to a similar operator proposed in the BSR parallel machine model (Broadcasting with Selective Reduction) [AG89]. The multiprefix provides the reduction, and also guarantees that all partial sums are computed in processor address order. This is similar to the segmented scan of the Connection Machine. Here, groups must be arranged into contiguous sites of a vector called "segments." Partial sums are computed in the site address order of the vector.

The **match** operation generalizes the specification of groups to the simple mechanism of keys. Keys are much more general than the site addresses used with **send** and **get**, the parallel variables used with the **fetch-and-op** or **multiprefix** primitives, or the segments associated with **scan** operations. **match** and **move** provide the functionality of the combining-**sends** and **get**, and also provide the reduction values associated with the **fetch-and-op** primitives and **scan** operations. However, they do not directly provide the partial sums of some of the primitives listed.

match and **move** may be implemented using any of the parallel primitives mentioned above by first associating each "group" of unique keys to the group required by the parallel primitive. The association may be through a unique variable, or a contiguous range of sites in a vector. Then, the parallel primitive supplied by the underlying hardware may be applied directly to implement a **move**. The mapping of groups to variables or sites may be achieved by sorting or hashing techniques. Thus, **match** isolates the programmer from the underlying architecture, but allows sufficient variation of implementation strategies for efficiency on a wide range of architectures.

1.6 Organization

This dissertation is divided into three distinct parts. Part I covers programming models. The first chapter of Part I introduces the parallel vector model and its relationship to the PRAM models. Following that, the **match** and **move** primitives and data types are introduced along with some simple examples. A detailed discussion follows that covers the complexity of the **match** operations as well as some simple optimizations that increase their efficiency.

Part II demonstrates the utility of **match** and **move** by presenting a large number of parallel algorithms written using this approach. It concentrates on algorithms for graphs, binary trees and sparse matrices. Most of the examples are well known parallel algorithms, but the presentation shows that they are easily expressed using **match** and **move** — each in less than one page of code. Furthermore, using the algorithmic analysis techniques of the earlier chapters of Part I, algorithms written at this higher level of abstraction often achieve the same efficiency measures as the best known EREW PRAM algorithms.

Part III covers the implementation of the **match** and **move** primitives on the Connection Machine CM-2 and the CRAY Y-MP in depth. On each machine, a variety of approaches are explored. While some approaches use existing programming libraries, some important new algorithms are developed. In particular, Part III presents a detailed analysis of parallel hashing for both machines and a novel implementation of a "multiprefix" operator for the CRAY Y-MP.

A concluding chapter compares the performance of a number of sparse matrix algorithms based on a highly optimized set of **match** and **move** operators to more traditional approaches. This research shows that an implementation based on **match** and **move** equals *or exceeds* the performance of even tightly optimized numerical subroutines. These results offer convincing arguments that a high-level approach to supercomputing using **match** and **move** need not suffer from a lack of efficiency.

Part I

Programming Models

Programming Models

The three chapters of Part I develop the programming model and notation presented in this dissertation. Chapter 2 introduces the Parallel Vector models. These models represent data as simple vectors and allow operations on entire vectors to be performed in parallel. Elementwise operators provide the basic means of computation for the entire family of models, while a wide variety of communication operators define the capabilities of specific models. Complexity measures for algorithms built on the Parallel Vector models are simple to compute but may also be translated into complexity measures for other parallel models.

Chapter 3 reviews the **match** and **move** operations and develops the notation used to write the algorithms presented here. While this notation is built on the Parallel Vector model, it expands the basic data types to include tuples for representing structured data. All variants of communication operators from the Parallel Vector models are subsumed by the functionality provided by **match** and **move**.

Chapter 4 discusses the complexity measures associated with the **match** and **move** operators and shows how they vary under different circumstances. Special cases of their use are identified for optimizations that ensure the algorithmic efficiency of programs written using **match** and **move**. Examples of these optimizations illustrate how propcrties of data vectors discovered at compile and run-time are used to ensure the efficiency of programs written in this manner.

Chapter 2

The Parallel Vector Model

The parallel vector model of computation builds on the standard RAM model by adding a separate vector memory and vector processing unit [Ble88]. Operations on vectors have the distinction of being performed in parallel. For example, the addition of two vectors can be performed for all elements of the result vector simultaneously. A standard RAM would iterate through each of the elements to calculate the same result. Because a single vector instruction may result in parallel execution, the parallel vector model is SPMD.

Fundamental to the vector model is the assumption that each vector may be of varying length. The amount of work each vector instruction performs is related directly to the length of its vector operands. To capture the complexity of algorithms written in the vector model, two complexity measures are used. The first is the "step" complexity, S, and measures the total number of vector instructions issued by an algorithm. The "work" complexity, W, is a related parameter that measures the total number of elements operated on over all instructions.

Elements of a vector are identified by their "site" index, which numbers the sites from 0 to $(n - 1)$ for an n element vector. The most primitive operations in the vector model are the "elementwise" operations. This class comprises all those instructions that are performed between elements with the same site index. Almost all simple scalar operations may be extended to parallel elementwise forms. Typical examples are mathematical operations and conditional expressions.

A	1	2	3	4	5	6	7	8	9	10
B	33	14	8	7	2	5	23	0	4	2

A+B	33	16	11	11	7	11	30	8	13	12
A>B ? A : B	32	14	8	7	5	6	23	8	9	10

The basic communication primitive of the vector model is the **permute** instruction, which rearranges the elements of a vector according to a permutation vector. This instruction allows only a simple one-to-one mapping of source sites to destination sites. The destination vector, to which the result is written, is given as the first argument to the **permute** instruction.

A	a	b	c	d	e	f	g	h	i	j	
p	5	2	9	7	1	0	3	6	4	8	
B	f	c	j	h	b	a	d	g	e	i	**permute**(B,A,p)

More general instructions are the **send** and **get** primitives, which are the exclusive-write and exclusive-read memory access methods of the model. The arguments to the **send** operation are a destination vector, a source vector, and an address vector that sends source data to the destination vector. Sites that are not referenced in the address vector receive no value and are not modified. The **get** operation retrieves data from the source to put it in the destination. The step complexity of each of these functions is $S = O(1)$ and their work complexity is proportional to the length of the active vector.

A	a	b	c	d	e	f	g	h	i	j	
p	3	4	1	7	5	9					
B	u	v	w	x	y	z					
G	d	e	b	h	f	j					**get**(G,A,p)
S		w		u	v	y		x		z	**send**(S,B,p)

There are also conditional forms of the **send** and **get** primitives called **cond-send** and **cond-get**. These take an additional flag vector that indicates active and inactive sites. For the **cond-send** operations, Inactive sites do not send their value to the destination. With **cond-get**, no value is retrieved from the source at inactive sites.

B	u	v	w	x	y	z					
p	3	4	1	7	5	9					
f	1	1	0	0	1	0					
S				u	v	y					**cond-send**(S,B,p,f)

A particularly powerful operation on vectors is the "scan" or "parallel prefix" operation. For any binary associative operator, $\oplus$, with an identity element **0**, and a vector v whose sites are referenced as $v[i]$, the $\oplus$-scan of v is defined for each site in v as

$$v[i] = \mathbf{0} \oplus v[0] \oplus v[1] \oplus \cdots \oplus v[i-1].$$

An example **add-scan** operation is shown below. Variants of the scan operations allow different combining functions (**add, mult, max, min, and, or**) on different data types (Boolean, Integer and Double). One other useful scan is the **copy-scan**; it copies the first value of the vector across all other sites.

A	4	3	2	2	8	6	7	8	9	10
B	0	4	7	9	11	19	25	32	40	49

add-scan(B,A)

Scan operators may accept another argument that designates *segments* within the vector. For each segment, the value of the scan includes only the sites since the beginning of the segment. There are a number of ways of representing segments, one of the clearest being a vector of bits with a **1** designating the start of a segment. Alternately, a separate vector giving the length of each segment, or the start position may be used. These forms are nearly interchangeable so that any of these forms that describe the segmentation of a vector is called a "segment descriptor." A representative segmented scan is shown below; the segment bits that are set are shown by arrows and the cleared bits are left blank.

segd	↓			↓				↓		
A	4	3	2	2	8	6	7	8	9	10
B	0	4	7	0	2	10	16	0	8	17

seg-add-scan(B,A,segd)

A vector of length n that enumerates its sites from 0 to $(n - 1)$ is called an "index vector." Such a vector can be created in one step by performing an **add-scan** on a vector whose sites have all been initialized to 1. Similarly, a vector that enumerates the sites of each segment is a segmented-index vector and can be created by using a **seg-add-scan** on a constant vector in one step. Two functions called **index** and **seg-index** are often used as shorthands for these two-step procedures.

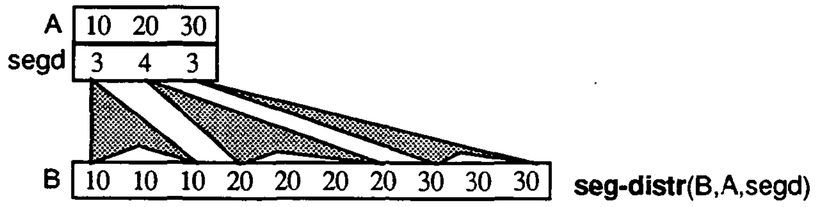
Segmented scans were first proposed by Schwartz [Sch80], but many more of their uses were realized by Guy Blelloch [Ble87]. Segmented scans have implementations nearly as fast as their non-segmented counterparts.

Closely related to the scan operations are the reduction operations. A regular reduction sums all of the elements of a vector to produce a single scalar result. Segmented versions of the reduction operations produce one element result for each segment. The same combining functions are available for the reductions as for the scans, because a reduction operation may be implemented through a scan.

segd	↓			↓				↓		
A	4	3	2	2	8	6	7	8	9	10
B	9	23	27							

seg-reduce(B,A,segd)

Values may also be copied over the segments with a segmented-distribute operation, **seg-distr**. Note that while the segmented reduction functions are related to the regular combining scan operations, this operation is related to a segmented version of the **copy-scan**. The example below illustrates a **seg-distr** operation with the segment-length type of segment-descriptor. It should be noted at this point that the two different representations of segmentation, start-bit and segment-length, can each be transformed into the other in just a few scan vector operations.



2.1 Relationship to PRAM Models

For a vector of n elements, all of the operators of the basic vector model have $S = O(1)$ step complexity and $W = O(n)$ work complexity measures associated with them. While the fact that the implementation of the scan operations involves a binary-tree combining operation convinces some that their time complexity should be logarithmic with respect to the size of the vector, in actual practice they are almost always as fast as (sometimes faster than) a single **send** operation. Blelloch argues that since unit time is based on a parallel memory reference, it is reasonable to ask what other class of operations execute as fast [Ble88, page 37]. This model is followed here, even though this view has not been universally adopted.

The *step* and *work* complexity measures are related to the PRAM model through the following relation. For an algorithm with step complexity S and work complexity W, the complexity of the algorithm is

$$t = O(W/p + S)$$

for a p-processor PRAM. This result is due to a simulation argument in which the vector algorithm is simulated on an EREW PRAM that supports scan operations [Ble88].

To translate these measures to a PRAM model that must simulate scans, the time complexity is

$$t = O(W/p + S \lg p).$$

All of our algorithms are designed with the unit-time scan in mind, so the former time complexity measure is assumed. However, skeptics can translate these figures by using the formula above. Because for most real algorithms, the problem size, n, and hence the amount of work, W, is many times larger larger than the number of processors, the $\lg p$ factor has a small effect.

2.1.1 Vector Algorithms are Adaptive

Parallel programs should be designed independent of a number of processors, p. To tie an algorithm to a particular number (or configuration) of processors would make such an algorithm inherently non-portable. Ideally, more processors should be used if they are available, giving better running times. A parallel algorithm with these properties is called "adaptive" [Akl89].

It is unlikely that the number of processors, p, will ever be a linear function of problem size, n. The cost of computer hardware will not keep up with the problem sizes people want to solve. As larger, faster computers have been developed in the past, scientists have set their sights on even more complex systems. Rather than ever meeting a need for processing power, the availability of more processors will simply stretch the maximum size problems that are attacked.

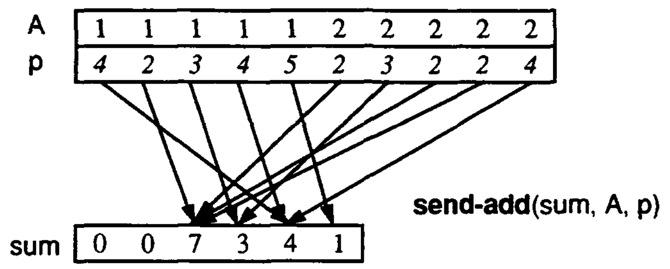
The vector model is adaptive and describes algorithms independent of the number of processors. Each vector instruction is simulated on the number of processors available, with each performing a share of the total work. Step and work complexity measures capture characteristics of an algorithm independent of the parameter p. Because the scan primitives are efficiently implemented on any number of processors, $p = o(n)$, an algorithm designed for the vector model will make use of whatever computing resources are available.

2.2 The CRCW Vector Model

The **send** and **get** primitives of the basic parallel vector model do not allow duplicate values in the address vector. However, many instructions exist on both real and theoretical machines that extend these operations to allow concurrent-read operations and combining-writes. Just as the PRAM models are broken into EREW, CREW, and CRCW categories, so are the vector models. Furthermore, the CRCW category is usually subdivided into models that further describe how concurrent write operations are handled.

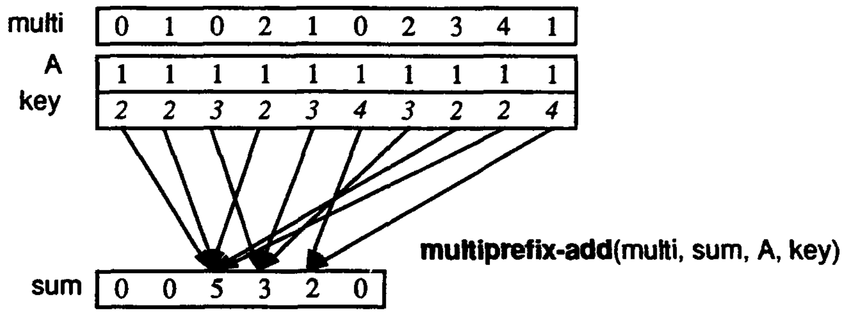
A **get** in the CRCW model allows multiple vector elements to reference the same site in another vector. This is the obvious concurrent-read extension of the **get** operator. More interesting are the variations possible for different concurrent-write operations.

Many different policies may be employed to handle collisions that occur in a **send**. The simplest such policy is that of colliding values an arbitrary one is chosen. This is the CRCW-ARB model. The CRAY Y-MP is an example of a machine that provides only this concurrent write operation. Another common policy extends the **send** operation to a family of operators that combines results sent to the same site with a binary associative combining function. Typical variants are **send-add** or **send-max** — essentially the same combining functions provided for the scan instructions. The Connection Machine CM-2 is an example of a machine that was designed with a combining network to support these instructions directly. These variants are all part of the CRCW-PLUS model, which supports combining. Because a CRCW-ARB machine cannot simulate a CRCW-PLUS efficiently, the CRCW-PLUS model is more powerful.



There is compelling evidence that if a parallel computer can support a combining-write, then without too much extra hardware a more powerful instruction can be provided [Ran87, AG89]. One such instruction is the "multiprefix," which is sometimes called a "keyed-scan." A similar instruction also appears in an asynchronous, shared-memory multiprocessor model with the "fetch-and-op" name. This complex instruction actually generalizes concurrent-read, combining-send and scan operations in one unified framework. The model that supports this operation is called the CRCW-MULTIPREFIX and it is stronger than CRCW-PLUS.

Instead of a segment descriptor, the multiprefix instruction takes a vector of small integer keys that are site indices in another vector. For each argument site with the same key, a prefix-scan is calculated in site address order that ignores sites with other keys. At the same time, the reduction of each group of sites with the same key is deposited in the site given by the key. An example illustrates the process for a **multiprefix-add** on a vector of 1s. This instruction takes arguments A and key and computes results multi and sum.



The operation of the multiprefix operator is most easily understood by tracing the scan operation for a single key: the 2 for example. The sites with this key receive the **add-scan** values $0, 1, 2, 3, 4$, while reduction site referenced by key receives the total sum, 5. The other keys are handled similarly. Notice too that the key vector really performs the role of an address vector and the sum vector the role of the destination.

As mentioned earlier, the multiprefix operator generalizes many other parallel primitives. If all keys are the same, then a simple scan operation is provided. If equal keys are contiguous, then the result is a segmented scan. A concurrent-read is implemented by using a multiprefix-add with a value of 0 as the initial value in the site of each key. All referencing sites then receive a copy of the value. Finally, a combining-send operation is provided directly when the multiprefix values are ignored.

Many parallel algorithms make use of some of these more exotic instructions of the CRCW vector model. Because they tend to be rather machine specific, algorithms written using them are generally not portable. However, if they are available on a given machine, a shrewd programmer will make use of them when they are appropriate.

Chapter 3

Vector Notation for match and move

This chapter introduces the basic data types and notation used to describe algorithms in our model. Most of this notation could be used to describe algorithms in the basic parallel vector models, but it will be especially useful for algorithms using **match** and **move**. While not quite a programming language, the notation presented here will be used as the pseudocode to describe our algorithms later. This chapter begins by introducing the basic scalar types and expressions and then extends them to vector types. Further details of the expression syntax may be found in Appendix A.

3.1 Scalar Types

Two types of values are distinguished: scalar and vector. Scalar values are of type Boolean, Integer, Double (precision floating point), or are a composite Tuple type. The first three atomic types are familiar, while the Tuple type serves the purpose of records or structures. A tuple is a fixed-length ordered collection of values, possibly of different types. The members of a tuple may be any of the atomic types, or may be a nested tuple. The grouping of values into a tuple is indicated by enclosing a sequence of values in brackets.

```
tup = <true, 13, 14.0, 2>;
```

The members in a tuple are identified by position only, and we call the left-most member the "most-significant" member and the right-most the "least-significant." To extract members of a tuple, an aggregate assignment is made to simple scalar variables. We call this operation tuple "dissection."

```
<a, b, c, d> = tup;
```

3.2 Vector Sets

A vector is a fixed length, ordered collection of values of a homogeneous type. Tuples, in contrast, may contain heterogeneous types. The elements of a vector are placed at distinct "sites" each of which is numbered from 0 to $(n - 1)$ for a vector of n elements. The number of each site is called its "index." A simple vector may be created by enclosing scalar values in brackets.

```
vect = [10, 11, 18, 23, 25];
```

In many algorithms it is natural to group together vectors of the same length when they store different quantities about the same collection of objects. For example, a collection of n edges in a graph would have head and tail values. While these could be stored in two unrelated vectors, a grouping of the vectors helps to clarify their purpose. A group of related vectors is called a "vector set" and all have the same length. Members of the same vector set are preceded by a common prefix and the constituent vectors of the set are called its "fields."

```
e.tail = [1, 2, 3, 4, 5];
e.head = [4, 3, 2, 5, 1];
```

Because the fields of a vector set are the same length, elements with the same index are at the same "site." (As an aside, vector sets serve almost the same purpose as Sabot's Paralations.)

A field that contains the index of each site is called the "index field" of a vector set and serves a special purpose. It is available in all vector sets with the field name #. The # field is assigned at the time of creation of the vector set and may not be reassigned. For example, the **new** function creates a new vector set and returns its index field.

```
e.# = new(size);
```

Other functions that create vector sets may return other fields in the set, but the index field is always available by referencing #. The index field plays a special role in many algorithms.

One way in which new fields are added to a vector set is by elementwise operations between existing fields. The new field is of the appropriate type for the result of the operation.

```
e.sum = e.a + e.b;
```

The other way fields are added is by a vector operation that creates a new field. All of the **move** operations are examples of such an operations. Most operations return fields in existing vector sets, although some return fields in new vector sets. The prefix notation helps make this explicit and keeps track of related fields.

3.3 Elementwise Operations

This notation extends all of the simple scalar operators of the "C" programming language to be elementwise operators on vectors. These include the regular arithmetic and logical operators (+, −, × , &, |, etc) the assignment operators (=, +=, &=, |=, etc.) and the conditional operator (? :). (Notice that we use × for multiplication instead of an asterisk.) Also provided are the operators **max** and **min**, even though "C" does not provide these directly. When expressions using these operators involve fields in the same vector set, an elementwise parallel application of the operators is implied. For example, the following code fragment selects the maximum between each pair of elements of two vectors.

```
e.maximum = (e.a > e.b) ? e.a : e.b;
```

A notational convenience promotes a scalar to a vector of constants when appropriate. In any elementwise expression involving a field and a scalar, the scalar value is replicated for all elements.

```
e.val = e.a + 3;
```

Elementwise operations may never be applied to fields belonging to two different vector sets.

Tuple creation is a scalar operation that may be extended to an elementwise operation as well. By enclosing the field names in angle brackets, a new field is created with a composite, tuple value at each site.

```
e.tup = e.<tail, head, val>;
```

Vector tuples may be dissected into their constituent fields by assignment to named fields. Because tuple members are ordered, the assignment is positional.

```
e.<t,h,v> = e.tup;
```

Tuples are used to treat composite values as a single entity, and to simplify the presentation of code. It is easier to pass a single vector of tuples to a function than to break it apart into its constituent fields.

Elementwise operations are so common that a proliferation of prefix strings sometimes obscures the desired computation. An **elwise** block can be used to name the vector set from which the field names are taken.

```
elwise(e) {
    val = a + 3;
    tup = <tail, head, val>;
}
```

Fuunction calls in an **elwise** block may not introduce nested parallelism, but indicate only that the function is applied elementwise to the values of the argument vectors. (A similar **elwise** construct was originally introduced in Paralation Lisp [Sab88a]. Notice that the use of this form is similar to the "with" statement in Pascal where all of the names refer to members of the same record structure.) Each statement in the block is a separate expression that produces a field in the same vector set. The choice of one of the two elementwise forms is based purely on the aesthetic values of the programmer.

3.4 The null value

The atomic types are extended to include the value **null**. This is not so much a value as an indicator of the absence of a value. A Tuple type has the **null** value if all of its constituent members do. The **null** value has a special role with the **match** and **move** operators, but we introduce it here to discuss its elementwise characteristics.

Any arithmetic operation involving **null** produces **null** as a result. **null** may be tested for equality, however, and may be the result of a computation such as a conditional operator. The following example produces a new field with all instances of val that are too big replaced with **null**.

```
e.clip = e.val > 100 ? null : e.val;
```

A special syntax has been created for the assignment operators that disables assignment if the right-hand value is **null**. An assignment operator preceded by a "?" indicates this special operation. In the following, sites with the **null** value have no effect on e.val.

```
e.val ?= e.clip;
e.val ?+= e.clip;
```

This shorthand is simply an abbreviation for the following scalar operation, but it has a special semantics that avoids evaluating its right argument twice.

```
DEFINE(a ?= b) → (b==null ? a : a = b)
```

In this notation, these conditional assignments are treated as a scalar operators that have an elementwise interpretation when applied to two fields. The full significance of the **null** value and the conditional assignment operators will become apparent later.

3.5 match

Elementwise expressions compute new values for each site in a vector set based on values at that same site. **match** and **move** allow values to be transferred between sites, and provide a simple means to combine values from different sites. Both **match** and **move** have been described earlier, they are re-introduced here using vector-set terminology.

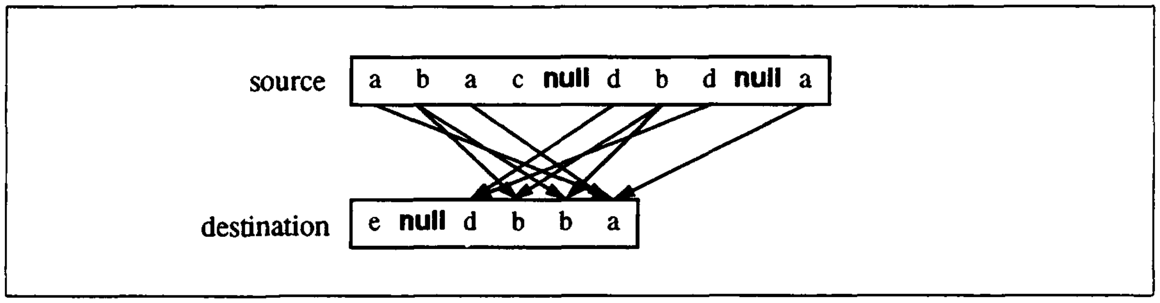


Figure 3.1: **null** keys match no others

The **match** operator uses two fields of "keys" to create a "mapping." For two fields called the "source" and "destination" a communication path is indicated from sites in the source vector set to sites of the destination vector set with the same key value. The mapping is a special data type that encapsulates the communication paths described between the fields of the source and destination.

```
M = match(destination.key, source.key);
```

Syntactically, in keeping with the manner in which an assignment statement indicates data movement from right-to-left, the **match** syntax mimics the same directional behavior.

The **null** value has a special meaning to **match**; it is a key value that matches nothing. Even **null** keys do not match one another. This is the mechanism used to exclude sites from a particular **match** operation. An example **match** involving **null** keys appears in Figure 3.1. The mapping created completely bypasses these sites.

Any type of field may be used in a **match** operation, but the source and destination must be of the same type. Keys of the simple atomic types (Boolean, Integer, Double) match if they test **true** for equality. Keys of a Tuple type match if each of their members match. Composite keys represented as Tuples provide a means to match sites based on a number of criteria at once.

3.5.1 Mappings do not capture all possible patterns

A mapping describes a gathering of the source and destination sites into "groups." Each group is a collection of source and destination sites with the same key. For this reason, the mappings created by **match** capture only a small subset of the possible communications patterns from source sites to destination sites. For any group of source sites that map to a destination site, it is not possible for a subset of those sources to map to another destination site unless all of them do. These characteristics follow directly from the fact that **match** is based on equality, and that it links all sites with equal keys.

Because of this grouping property, a better pictorial representation of a mapping gathers source and destination sites together if they are in the same group. Each group in Figure 3.2 is represented by a circle and its members are connected by arcs. Group members are further

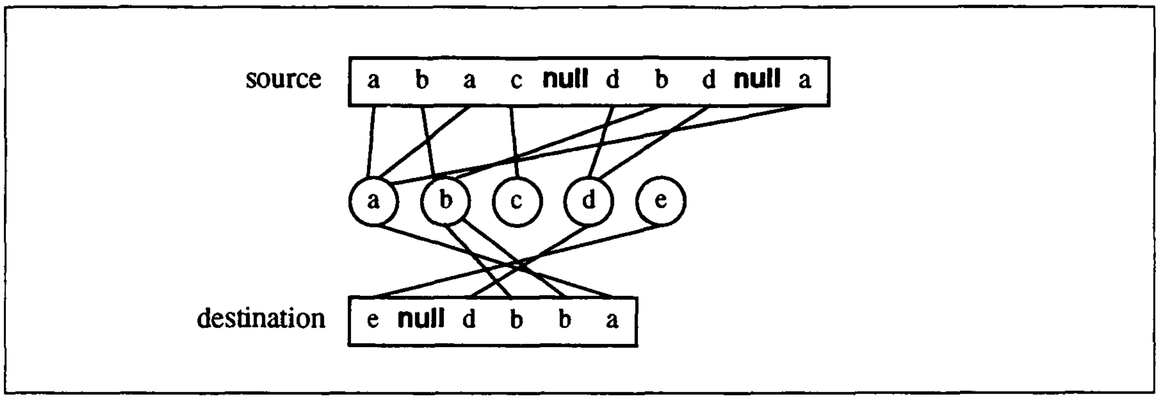


Figure 3.2: A representation of a mapping that illustrates groups

differentiated as to whether they are a source or destination. As an aside, this representation also alludes to an efficient implementation for operations that **move** data through a mapping by showing that the total number of communication paths need not exceed the total number of sites involved.

3.6 move

The **match** and **move** operators clearly distinguish between the description of a communication pattern as a mapping, and the movement of data over that pattern with a **move** operation. When multiple source values are mapped to the same destination, variants of **move** provide for different collision handling policies. This section will explain each of the **move** operations and how they may be applied to the different data types.

The **move-arb** operation may be applied to any of the data types; it simply moves copies of the source values to the destination sites. If there are collisions, an arbitrary one of the source values is chosen. Unlike Sabot's `#'arb` combining function, this operator does guarantee that each destination of the same group receives the same value. The following example demonstrates a possible outcome of a **move-arb** operation. In the examples, letters used for the keys are a symbolic value of some unknown type. The last line is a comment which describes the result of the computation. Comment lines begin with string "//."

```
s.key = [a, b, c, a, b, d, c];
s.val = [1, 2, 3, 4, 5, 6, 7];
d.key = [b, b, a, a, c, c, a, a];

M = match(d.key, s.key);
d.val = move-arb(M, s.val);
//result = [5, 5, 4, 4, 3, 3, 4, 4]
```

The **move-add** operation may be applied to numeric values (Integer or Double). Values destined for the same destination site are added together.

```
s.key = [a, b, c, a, b, d, c];
s.val = [1, 2, 3, 4, 5, 6, 7];
d.key = [b, b, a, a, c, c, a, a];

M = match(d.key, s.key);
d.val = move-add(M, s.val);
//result = [7, 7, 5, 5, 7, 7, 10, 10, 5, 5]
```

The **move-max** and **move-min** operations may be applied to fields of *any* type. For numeric fields they have the usual effect. On Boolean values **true** is greater than **false** and they have the same effect as **move-or** and **move-and**, respectively. Tuple values are treated as composites, and the comparison proceeds from the most-significant field to the least-significant. For example, the tuple <5,3,3> is greater than <5,3,2>. Similarly, the tuple <4,6,6> is greater than <3,7,7> for these comparisons.

```
s.key = [a, a, a, a, a, b, b, b];
s.val1 = [1, 2, 3, 4, 5, 2, 2, 2];
s.val2 = [3, 3, 3, 3, 3, 1, 2, 3];
d.key = [a, b, a, b];

M = match(d.key, s.key);
d.val = move-max(M, s.<val1,val2>);
//result = [<5,3>, <2,3>, <5,3>, <2,3>]
```

While this is not the place to discuss implementation, many readers will be wondering how comparisons on tuples can be implemented efficiently. In most cases, the members of a tuple can be packed into a single machine word before the combining operation is applied. When this is not the case, other methods may be employed. These are considered in the last part of this dissertation, which treats implementation in detail.

3.6.1 null values and non-matching sites

As shown earlier, mappings need not include all source and destination sites. A site may not be matched for either of two reasons. First, the key at that site may not have had a matching key on the other side. Secondly, the key may have been **null**. A **move** operation produces **null** values at those destination sites that were not matched when the mapping was produced.

In the following example, two **null** values are produced. The first is because of a key "e" that did not match a source site. The second is because of a **null** key that could not match a source site by definition of the meaning of **null**.

```
s.key = [a, b, c, a, b, d, c];
s.val = [1, 2, 3, 4, 5, 6, 7];
d.key = [a, e, b, null, a];

M = match(d.key, s.key);
d.val = move-add(M, s.val);
//result = [5, null, 7, null, 5]
```

Why use null values?

These **null** values are important so that the result of each **move** operation is a new vector field. This allows the **move** to produce an entire vector as a result, instead of modifying a destination vector in a side-effecting manner. It also concretely expresses the complexity of the operation in that the computation includes all of the destination sites. It isn't any more efficient to write a program that uses just a few sites of a vector!

A common idiom in this notation supplies default values in a vector before a **move** operation. Remember that the ?= operator is a conditional assignment that only replaces values with non-nulls. The following code fragment uses the same vectors as before but initializes a vector to the default value 1 before overwriting it with the result of a **move-add**.

```
s.key = [a, b, c, a, b, d, c];
s.val = [1, 2, 3, 4, 5, 6, 7];
d.key = [a, e, b, null, a];

//give destination field a default value
d.val = 1;
M = match(d.key, s.key);
d.val ?= move-add(M, s.val);
//result = [5, 1, 2, 1, 5]
```

The ?= operator ensures that **null** values do not overwrite the default.

3.6.2 move-join

The introduction showed how the **move-join** operation creates a new site for each path indicated in a mapping (see Figure 1.2). Another way of viewing this operation is as one that "concatenates" the values arriving at a destination and places them in adjacent sites in a new vector. When illustrating the result of a **move-join**, the new sites fan-out from the destination site, with one new site for each matching source.

As opposed to the other **move** operators, **move-join** requires a mapping and *two* other arguments: a field from the source, and a field from the destination. The result of **move-join** is a tuple field in a new vector set that has one site for each path of the mapping. The members of the tuple include the members of the destination field in the most-significant positions, and the members of the source field in the least-significant positions. This arrangement allows the corresponding field names on the left and right of an assignment to correspond in a simple left-to-right manner.

```
M = match(s.key, d.key);
new.<ff1,ff2,ff3,ff4> = move-join(M, s.<f1,f2>, d.<f3,f4>);
```

Many things are happening in the **move-join** statement. First, both of the arguments are elementwise expressions involving fields that produce a two member tuple as a result. The **move-join** operation produces a tuple field as a result with four members, that is then dissected into four fields by the assignment statement. The values of the members, in most- to least-significant order come from the source and then the destination.

Non-matched sites produce no new sites in a vector set produced by **move-join** because a destination site that is not matched produces 0 sites in the new vector. Similarly, a source site that is not matched does not add sites for any of the destination sites. Figure 3.3 illustrates the fanning out behavior for non-matched sites. **move-join** is often used in this manner to create new vector sets based on a correspondence between two existing vector sets.

3.7 Complexity Measures

The complexity measures used to describe algorithms written using **match** and **move** are those of the parallel vector model. All elementwise operations have an $S = O(1)$ step complexity and $W = O(n)$ work complexity for a vector of n elements. Elementwise operations include all of the regular arithmetic operations, tuple creation and dissection, assignment operations, and conditional assignment operations.

All of the **move** variants have an $S = O(1)$ step complexity measure as well. This measure is defended when the implementation of the **move** operation is discussed. For a regular combining **move**, the work complexity is $W = O(s + d)$ where s and d are the lengths of the source and destination vector sets. The work complexity of a **move-join** is $W = O(s + d + j)$ where j is the number of sites created in the new vector as the result of the **move-join**.

In the worst case, the complexity measures of a **match** operation are $S = O(\lg n)$ and $W = O(n \lg n)$ where $n = s + d$, the sum of the lengths of the source and destination. In many cases, these measures over-estimate the complexity of creating a mapping. The next chapter treats this issue in detail and discusses how optimizations are employed in a systematic manner.

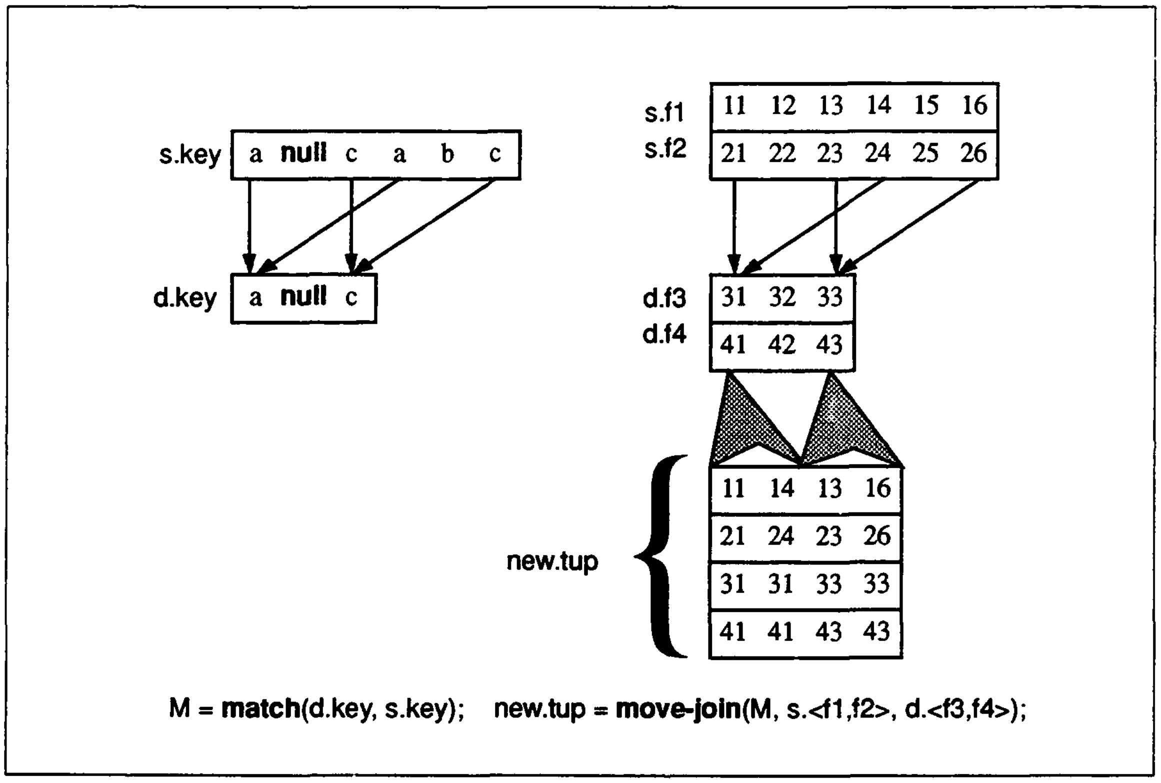


Figure 3.3: An example **move-join** with **null** sites. Destination and source sites that don't match are essentially "compressed" out in the expansion of the joined vector set.

Chapter 4

The Complexity of match

The sole method for describing a communication pattern is through a mapping built with **match**. Because of the generality of the **match** operator, a wide variety of mappings can be described. It is easy to place bounds on the number of parallel steps required to implement a worst-case general-purpose **match**, however, some instances may employ optimizations. In many cases the opportunity to use an optimization is recognized automatically with only a small amount of extra bookkeeping. However, in other situations, the programmer must direct optimizations by using directives and hints in the form of **pragma** statements.

Different complexity measures for one operator

Different combinations of keys can lead to radically different mappings. It would be misleading to treat all mappings the same, because it is natural that a parallel computer will perform some communications operations better than others. A simple classification of mapping types captures this fact and exposes it to the programmer.

While it may seem desirable to have a single complexity measure associated with each operator, it is the sheer generality of **match** that makes that assumption unrealistic. The operations performed by **match** attempt to "understand" a communication pattern described by keys. Often, **match** can recognize the pattern described in much less time than the most general case. These special cases are easily recognized when given the right set of inference rules for fields and mappings. With the knowledge of these rules, a programmer is able to understand why some mappings are more efficient than others and is better able to predict their performance from the complexity measures derived.

A Variety of Architectures

Our algorithms are designed for the parallel vector model, but many parallel computers offer capabilities that are not directly captured in this model. While the implementation of **match** guarantees upper-bounds on the complexity of its implementation using the primitives of the vector model, other machine features are employed when possible. For instance, a mapping that represents a concurrent-read is simulated in the parallel vector model by using a number **send** and **scan** operations. If an algorithm with such a mapping is ported to a machine that supports this operation directly, a direct concurrent-read might possibly perform better.

The decision of an implementation strategy for each type of mapping is best left to the **match** machinery. An aggressive implementation of **match** and **move** should take into account many different aspects of the underlying computer. By placing these decisions in the software layer underlying **match**, the programmer is reasonably assured that a suitable approach will be used for each mapping. In this manner a novice programmer using **match** and **move** automatically inherits the expertise of someone more knowledgable about the low-level details of a machine, while the use of **pragma** directives allows the expert direct control over the implementation strategy.

4.1 The Implementation of a General-Purpose match

A **match** operation may be applied to two fields with elements of any type. These may be Boolean, Integer, Double or Tuple fields. A mapping produced with **match** encapsulates a communication pattern that groups source and destination sites, where a group is identified by sites with equal keys. The general implementation strategy associates each group of keys with a combining operation supported by the underlying parallel computer.

In the parallel vector model, a simple **match** and **move** strategy based on sorting and segmented scans can be devised. The idea behind constructing a mapping through sorting is that equal keys should be brought near each other by a sorting operation. This type of mapping associates the groups specified by keys with a segment in a vector. The combining function appropriate for this type of group is the segmented-**scan**.

The steps involved in building a mapping are illustrated in Figure 4.1. First, the keys of the source and destination vector are concatenated into one vector called the "agent" along with pointers back to their sites of origin. These site addresses are called the "home" pointers for the keys. After sorting the agent elements by their key value, the agents are divided into segments so that each group of equal keys is in one segment. The identification of segments is done in one parallel step by comparing each key to its left neighbor in the sorted agent vector. Finally, the indices of the new sites in the agent vector are sent back to the home site of each key. The agent vector along with the pointers from keys to agent sites becomes the data structure of the mapping.

A **move** operation using this type of mapping is illustrated in Figure 4.2. Three steps complete the **move**. First, the source values are sent to the agent. Next, a single **scan** combines the values of the source agents and distributes the sum across the destination elements. The last step sends

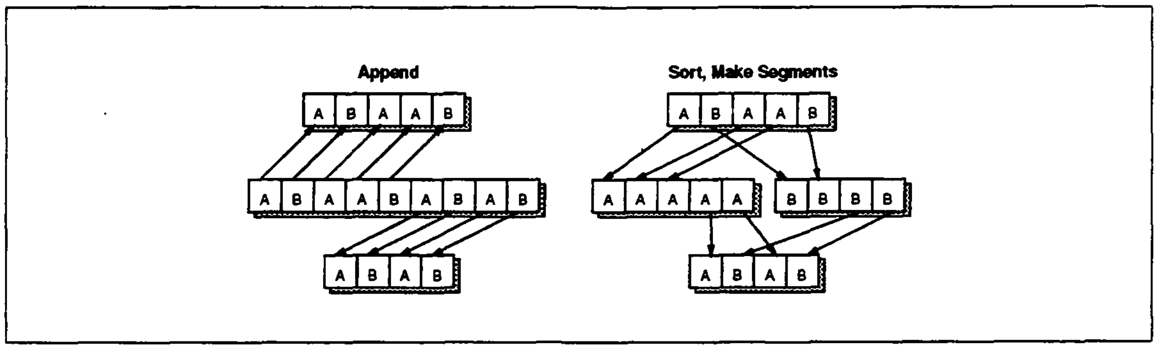


Figure 4.1: A mapping may be created by using sorting. In the first step, agents are appended from the source and destination vectors into a new vector. After sorting the agents, they are divided into groups by comparing each key to its neighbors.

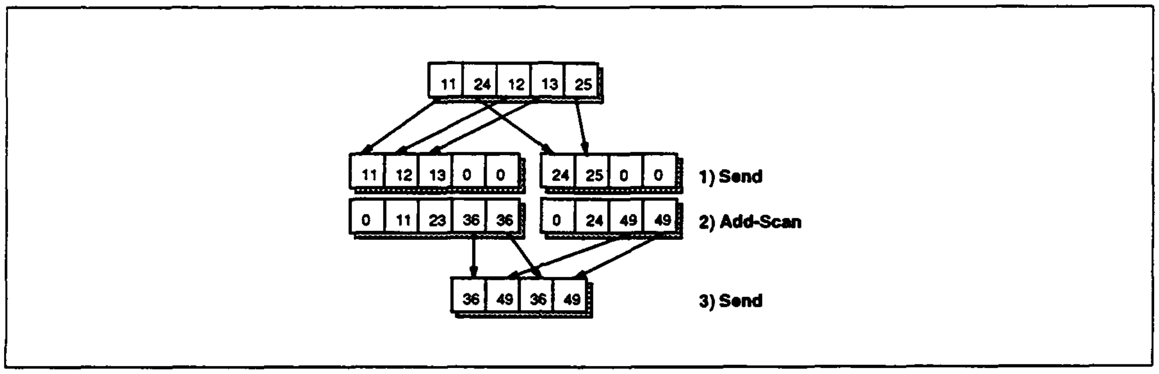


Figure 4.2: Using a mapping created by sorting to perform a **move-add**. The first step sends the source values to the agent vector, where they are combined with a segmented **add-scan**. The final **send** forwards the results to their destinations.

the sums in the destination agents to their final sites The operations involved in each of the three steps are unit step ($S = O(1)$) operations in the vector model. If the sum of the lengths of the source and destination vectors $n = S + D$, then the work complexity of **move** is $W = O(n)$.

The step that dominates the **match** procedure above is, of course, the sorting step. For the parallel vector model, two bounds on sorting exist. For sorting n general keys based on comparison, the Quicksort algorithm obtains $S = O(\lg n)$ and $W = O(n \lg n)$ expected step and complexity bounds [Ble88, page 47]. Alternatively, using the split-radix sort, a vector of r-bit values is sorted in $S = O(r)$ steps with $W = O(rn)$ work [Ble88, page 42]. Since n distinct values require at least $(\lg n)$ bits to be represented, these bounds are closely related as problem sizes grow.

In general, a **match** operation requires $S = O(\lg n)$ steps and $W = O(n \lg n)$ work. However, with knowledge of the range of values sorted, better bounds are obtained for the performance of the operation. In particular, when the range of key values is a constant for all problem sizes, a "fixed-range" **match** is indicated and has $S = O(1)$ step and $W = O(n)$ work complexities. Such a situation occurs most often when creating a mapping on Boolean keys. Surprisingly, many algorithms can be constructed by performing a **match** on Boolean keys.

For example, the "reduction" of a vector of values is a scalar value that is the sum of all the elements with respect to some binary associative operator. Using **match** and **move**, the **reduce-add** operation is defined to sum all of the elements of a vector as shown below. (In our algorithms, the result of the computation is assigned to the function name at the end of the code.)

```
reduce-add(v.val) {
    v.key = true;                              //vector of constants
    singleton.key = [true];
    M = match(singleton.key, v.key);           //all sites match
    singleton.val = move-add(M, v.val);
    reduce-add = singleton.val[0];             //extract reduction value
}
```

While this may seem convoluted at first, it is really quite simple. First, a vector of Boolean **true** values is distributed across the set v in field v.key. Then, a vector of one element with the value **true** is created. Each of these operations is a simple $S = O(1)$ step operation. The **match** statement maps all elements of v to the same site in the one-element vector. Because the keys are Boolean values, this is a fixed-range **match** and requires only $S = O(1)$ steps. All of the values are then added together using **move-add**.

The step and work complexities of this algorithm are $S = O(1)$ and $W = O(n)$. Notice that in the vector model, a single scan suffices to calculate the reduction and results in the same complexity measures. Thus, our algorithm correctly captures the complexity of this simple operation in the vector model, while it avoids dictating an implementation strategy for calculating the result.

4.2 Inference Rules and Optimizations

Many of the mappings created in an algorithm written using **match** and **move** are much simpler than the general many-to-many mapping. For example, in any **match** from an index field to an integer field, a simple concurrent-read is indicated.[1] An aggressive implementation of the **match** operation must recognize this situation and exploit it when it can.

This section presents an inference system about fields and mappings that attempts to preserve information about the contents of these structures. A number of important optimizations are recognized after introducing only a few concepts. With these optimizations, it is possible to express some typical parallel kernels at a high level without sacrificing efficiency.

Two points will be made in this section. If special properties are known about the source and destination key fields used to create a mapping, then with no increase in complexity it is

[1] In[Gol89], Kenneth Goldman also recognized index vectors as a special case, but in a much less general framework than is presented here.

possible to deduce facts about the type of the mapping. Secondly, when using a mapping with a known type and a field with known properties in a **move** operation, it is often possible to deduce characteristics of the result field. With this additional knowledge of mappings and field contents, it is possible to apply some important optimization strategies.

The optimizations presented here concern savings in asymptotic step complexity only. Constant factors must be taken into account as well because complexity measures alone can hide many anomalies of a parallel computer under big-O notation. Constant factor optimizations require a detailed understanding of implementation strategies. For this reason, their discussion is deferred until Part III.

Field Properties

Fields may have any number of properties associated with them. A property is a flag value that describes characteristics of the values of a field. If a field has the "Unique" property, then all of its values are different, except that there may be any number of **null** values. (Remember that a **null** value matches no other key.) An "Enumerable" field is one in which the values of the field may be translated into integers in some range $[0..R)$ through a simple known elementwise ($S = O(1)$) translation function. The value of R is a bound on the number of distinct values in the vector.

If a field has "Sorted" property, then its values are strictly non-decreasing. Another property is the "NoNulls" property and indicates that there are no **null** values in the field. Fields may be created with these properties, or they may be detected at run-time. For example, all index fields are known to have the Unique, Enumerable, Sorted and NoNull properties at all times. For other fields, these properties may not be so easily detected.

Mapping Types

Mappings created by **match** can fall into one of three special forms, or can be purely general. A mapping about which nothing is known is called a "Full" mapping. A mapping in which each source site matches at most one destination site and each destination matches at most one source is called "EXclusive" because a **move** according to it can be implemented with either an exclusive-read or an exclusive-write. Sites that match nothing are also permitted in this type of mapping.

The only other two special types are the "CR" and "CW" mappings. These represent mappings for which a simple **move** can be implemented with a concurrent-read or a concurrent-write operation, respectively. An implementation for a machine that supports these operations directly can make use of this information to speed execution. A CR mapping is recognized when no destination site matches more than one source site. Similarly, a CW mapping is recognized when no source site matches more than one destination. This characterization also allows non-matching sites in the mapping.

4.2.1 Deducing Mapping Types from Source and Destination Properties

Using the properties of the fields involved in a **match** operation, it is often possible to directly deduce the type of the resulting mapping. With the knowledge of the type of the result mapping it may also be possible to use a reduced complexity algorithm to build the mapping. For this reason, it is important to recognize mapping types before they are created.

The Uniqueness properties of the source and destination key vectors are sufficient to determine whether the mapping created is one of the special types. Table 4.1 summarizes the deduction rules for mappings of a special type. The entries in the table follow directly from the definition of Uniqeness and the special mapping types.

Table 4.1: The resulting mapping type based on Uniqueness properties. This one property alone is sufficient to recognize a simplified mapping.

	Source	
Destination	Unique	not-Unique
Unique	EX	CW
not-Unique	CR	Full

4.2.2 Deducing Field Properties from move Operations

The type of the mapping and properties of the argument fields are used to deduce the properties of a result field for a **move**. While little can be ensured for the result field for the arithmetic combining functions, for **move-arb** and **move-join** the Enumerable and Unique properties may be detected. For instance, passing any Enumerable field through a mapping of any type with **move-arb** or **move-join** produces an Enumerable field as a result with the same range, R, because the same values appear in the result.

The rules for the Uniqueness property are a little more complicated. The justification for these rules follows directly from the definition of the Unique property and the mapping types. These rules are summarized in Tables 4.2 and 4.3.

Because a **move** operation may be used on tuples, each of the entries concerns a single field of the result produced by a single argument field. The entries in the **move-arb** table describe the Uniqueness property of the result based on an argument field to **move-arb** with the mapping type indicated. The result of the **move-join** operator is a tuple whose fields come either from a source data field or a destination data field. For this reason, results computed from source or destination fields are described in separate columns.

Table 4.2: Deducing the Uniqueness property for **move-arb**. The Uniqueness property may be deduced about the result field of **move-arb** based on the type of the mapping and the Uniqueness of the argument.

move-arb

	Source Data	
Mapping Type	Unique	non-Unique
EX	Unique	X
CR	X	X
CW	Unique	X
Full	X	X

Table 4.3: Deducing the Uniqueness property for **move-join**. Because **move-join** combines fields of both the source and destination, each must be considered independently. However, the Uniqueness property may be detected for each at runtime.

move-join

	Source Data		Dest Data	
Mapping Type	Unique	non-Unique	Unique	non-Unique
EX	Unique	X	Unique	X
CR	X	X	Unique	X
CW	Unique	X	X	X
Full	X	X	X	X

A description of one case will clarify the rules for **move-join** in Table 4.3. A CW mapping is one in which each source matches zero or one destination sites. Because **move-join** creates cross-products, there is at *most* one result site for every source. When applying **move-join** to a Unique source field with such a mapping, then each source value is copied at *most* one time, yielding another Unique field.

4.3 Using Properties and Mapping Types to Optimize match

Some substantial optimizations are simple to implement by recognizing the few properties introduced. These optimizations take advantage of special circumstances, so they usually depend on properties of the source and destination fields as well as the type of the resulting mapping, if it is known. These rules are laid out in Table 4.4 along with the associated step and work complexities required to build the mapping.

Most of these optimizations make use of the Enumerable property of the source and destination keys. This property indicates that the key field may be converted to small integers in $O(1)$ steps. A typical example is an integer field that can be translated by adding the appropriate offset. The interpretation of "small" is based on the context of the use of the optimization rule.

Table 4.4: Situations for an optimized **match**. By making use of information known about the source and destination, and possibly the type of the resulting mapping, tighter bounds are placed on the complexity of implementing the **match** operation.

	Source/Dest Properties	Mapping Type?	Condition	Step	Work
1	Enumerable - range R		$R = O(n)$	$O(\lg R)$	$O(n \lg R)$
2	Enumerable - range R		$R = O(1)$	$O(1)$	$O(n)$
3	Enumerable - range R	EX	$R = O(n)$	$O(1)$	$O(n)$
4	(none)			$O(\lg n)$	$O(n \lg n)$

Rules 1 and 2 are really the same. Rule 1 applies if the range, R, is on the order of the length of the fields involved in the **match**. If it is, then the keys may be sorted with radix sorting in $S = O(\lg R)$ steps as mentioned earlier. The second rule merely points out the special case when the range is constant for all problem sizes. For example, it is 2 for a **match** involving Boolean fields.

Rule number 3 applies if the type of the result mapping is known to be EX before it is even created. The condition is that the range, R, is on the order of the number of elements involved in the **match**. In reality, it may be no larger than the largest size vector that the implementation can allocate. A typical example of a **match** that would apply this optimization is one involving two vectors of small integers that are both Unique.

The justification for this rule is that since the mapping is known to be EXclusive, it can be transformed into an exclusive-write operation of the vector model using the site addresses of the destination directly. The process involves sending the index of each destination site to the source site that matches the key. Because the mapping will be EX, only one such site can exist for each destination.

A temporary vector of length R is allocated and each of the source and destination keys are translated into small integers that directly reference the sites of this vector. In this way, equal keys of the source and destination can use sites in the temporary vector to communicate values.[2] First, each destination site sends its index to the temporary vector, where it is retrieved by its matching source. This does not complete the process however, as the source sites with no matching destination must be detected.

One approach first clears the temporary vector to **null** values, but since it may be very large this approach is not efficient. This inefficiency can be avoided by initializing the sites of the temporary vector through the following procedure. After the steps above, a **null** value is written to the temporary vector sites named by destination keys. The source sites then re-read the sites

[2]This process is actually a special case of hashing, where the hash function is guaranteed to produce no collisions. The use of hashing is discussed in detail in Chapter 9.

of the temporary vector. If the value read has changed from an index to a **null** value, then a source has indeed matched a destination. If the value does not change, or is **null** on both steps, then a source does not match a destination. It then changes its destination index to a **null** value, indicating no destination. Destination sites may detect if they do not match by a similar process.

4.4 Aggressively Seeking Properties and Types

If nothing special is known about the source or destination fields, it is still possible that the mapping indicated is a special type. This section answers the following question:

> If a full **match** must be performed anyway, what can be learned about the result mapping and the source and destination keys for the same amount of work?

By "work" is meant the same asymptotic step and work complexities required to perform the full **match**. Recognizing that if an algorithm computes a mapping

```
M = match(destination, source)
```

then the reverse mapping

```
Mbak = match(source, destination)
```

is computed with the same complexity measures. Furthermore, a simple (non-join) **move** operation is performed in unit time with $W = O(n)$ work complexity. The quantities of interest are the maximum number of source/destination sites in one group, and whether all non-null source/destination sites matched.

Two auxiliary functions are shown below. Given a mapping, the function **maxGroup** computes the maximum number of source sites in one group. The other function **allSitesMatch** returns **true** if all of the non-**null** destination sites in a mapping are matched. The step and work complexities of both of these functions are $S = O(1)$ and $W = O(n)$.

```
maxGroup (M,s,d) {
    d.groupsize = 0;
    d.groupsize ?= move-add(M, 1);
    maxGroup = reduce-max(d.groupsize);
}

allSitesMatch (M,s,d) {
    d.nonnull = (d != null);
    d.hasmatch = false;
    d.hasmatch ?= move-arb(M, true);
    d.mark = (!d.nonnull | d.hasmatch);
    allSitesMatch = reduce-and(d.mark);
}
```

Using these two functions, the mappings M and Mbak, and the original source and destination key vectors it is easy to compute four important quantities. They are:

- maxSourceGroup = **maxGroup**(M,source,dest)
  The maximum number of source keys in a group.
- maxDestGroup = **maxGroup**(Mbak,dest,source)
  The maximum number of destination keys in a group.
- allDestsMatch = **allSitesMatch**(M,source,dest)
  Whether or not all non-null destination keys have a matching source key.
- allSourcesMatch = **allSitesMatch**(Mbak,dest,source)
  Whether or not all non-null source keys have a matching destination key.

Now, with these pieces of information, the original mapping, M, may be classified even if nothing was originally known about the source and destination keys. The maximum group size values determine whether the mapping is simpler than a many-to-many mapping. Table 4.5 illustrates the simple decision process based on the sizes of the groups. The table is nearly identical to the one for mappings resulting from Unique fields. The justification for the resulting mapping types in the table follows directly from their definition.

Table 4.5: Recognizing simplified mapping types. The maximum sizes of source and destination groups are all that are needed to determine if the mapping is EXclusive, CR, CW or Full.

	MaxSourceGroup	
MaxDestGroup	1	> 1
1	EX	CW
> 1	CR	Full

With the knowledge of the mapping type and whether all of the source and destination sites matched, the following two rules may be applied to determine if the original source and destination key fields have the Unique property.

- If all non-null destination sites match, and the mapping is EX or CW, then the destination is Unique.
- If all non-null source sites match, and the mapping is EX or CR, then the source is Unique.

These rules are easily defended. The first one follows from the fact that an EX or CW mapping has no two destination sites in the same group. This implies that they are all unique but only if all of them actually matched, because equal destination keys that have no matching source could also produce an EX or CW mapping. A similar argument defends the second rule.

Note that the information about the type of the mapping and the Uniqueness property of the source and destination fields was computed with no increase in asymptotic complexity. In the real implementation of **match** however, the four quantities computed above are computed with nearly no extra *absolute* work. However, that analysis requires a detailed discussion of the implementation of **match** which is deferred until a later chapter.

The next section will show how the extra work of finding the type of a mapping can pay off in the long run. Combined with the inference rules, this simple optimization system can greatly enhance the performance of many real algorithms.

4.5 An Example of Optimization

Pointer doubling is a powerful algorithmic technique used in many parallel algorithms. For a vector V, whose values are interpreted as indices into V, a single pointer-doubling step computes the value

$$V(i) = V(V(i))$$

for each site of the vector. Using **match**, this operation is expressed as moving the value of each site to those other sites that reference it by site index.

```
double(v.i) {
    double = move-arb(match(v.i, v.#), v.i);
}
```

The mapping is from a Unique-Enumerable field (an index) to a non-Unique, but Enumerable field, which is recognized as a CR mapping. However, because the range, R, of the field is n, **match** still requires $S = O(\lg n)$ steps and $W = O(n \lg n)$ work in the vector model.

Pointer doubling can be used to distribute a value through a rooted tree in a logarithmic number of iterations. A forest of rooted trees is described as a directed graph in which every vertex has one successor (out-degree one), and every component has exactly one cycle of length one. The root of such a tree is the vertex that is its own predecessor. The doubling process can be used to transmit a value associated with the root to each of the vertices in its component. Most often the value transmitted is the index of the root so that the entire process labels each vertex with the index of its root in the component.

The general purpose doubling algorithm is shown in Algorithm 4.1. The argument fields are the graph pointers and the initial values of each vertex. An additional field called next is used to hold the pointers as they evolve. The **null** value is used as a sentinel to mark a root value. In the initialization step, roots are recognized as those vertices that point to themselves and are given a **null** value in their next field. Once a vertex finds the sentinel value, it knows that it has found the

```
double* (v.<ptr,val>) {
    v.next = (v.ptr == v.#) ? null : v.ptr;
    lgN = lg(length(v));
    for (i = 0; i < lgN; i++) {
        M = match(v.next, v.#);
        v.next = move-arb(M, v.next);
        v.val ?= move-arb(M, v.val);
    }
    double* = v.val;
}
```

Algorithm 4.1: The basic pointer doubling algorithm. Each doubling step transmits the root value to vertices twice as far away as the preceding step.

value associated with the root and participates in no later matches because its next field becomes **null**.

Figure 4.3 shows an initial rooted tree and the evolution of its next pointers using the procedure **double*** ("double-star") shown. The sentinel value is represented as a hollow circle. As each vertex receives the sentinel value, its next field becomes **null** and it participates in no further **match** steps. The conditional assignment operator (?=) is used to ensure that the val field of vertices that have already found the sentinel are not overwritten. For an n vertex graph, after $(\lg n)$ iterations, the root value will be distributed to each of the vertices. Because all of the mappings are CR, the complexities of this procedure are $S = O(\lg^2 n)$ and $W = O(n \lg^2 n)$.

4.5.1 List Doubling

A special case of pointer doubling occurs when the pointers describe a rooted linear linked list. All vertices of this graph have an in-degree and out-degree of 1, except the root which is also its own predecessor. Through careful algorithm design, a list doubling procedure can ensure exclusive-read access. By using the inference rules given in this chapter, the **match** and **move** operators can discover this fact with no advice from the programmer, yielding an $S = (\lg n)$ step complexity algorithm for the special case of a linked list for the same **double*** function.

Figure 4.4 unrolls and annotates the statements of the **double*** function when the next field describes a linked list. Initially, no properties are known about the next field. However, if it describes a simple linked list, then after the first **match** two things will be discovered. First, the mapping will be recognized as an EXclusive type. Then, with the knowledge that the mapping turned out to be EXclusive, and the source was Unique (an Index field), the next field will be marked with the Unique property. This information is obtained at the expense of a full **match**, or $S = O(\lg n)$ steps and $W = O(n \lg n)$ work.

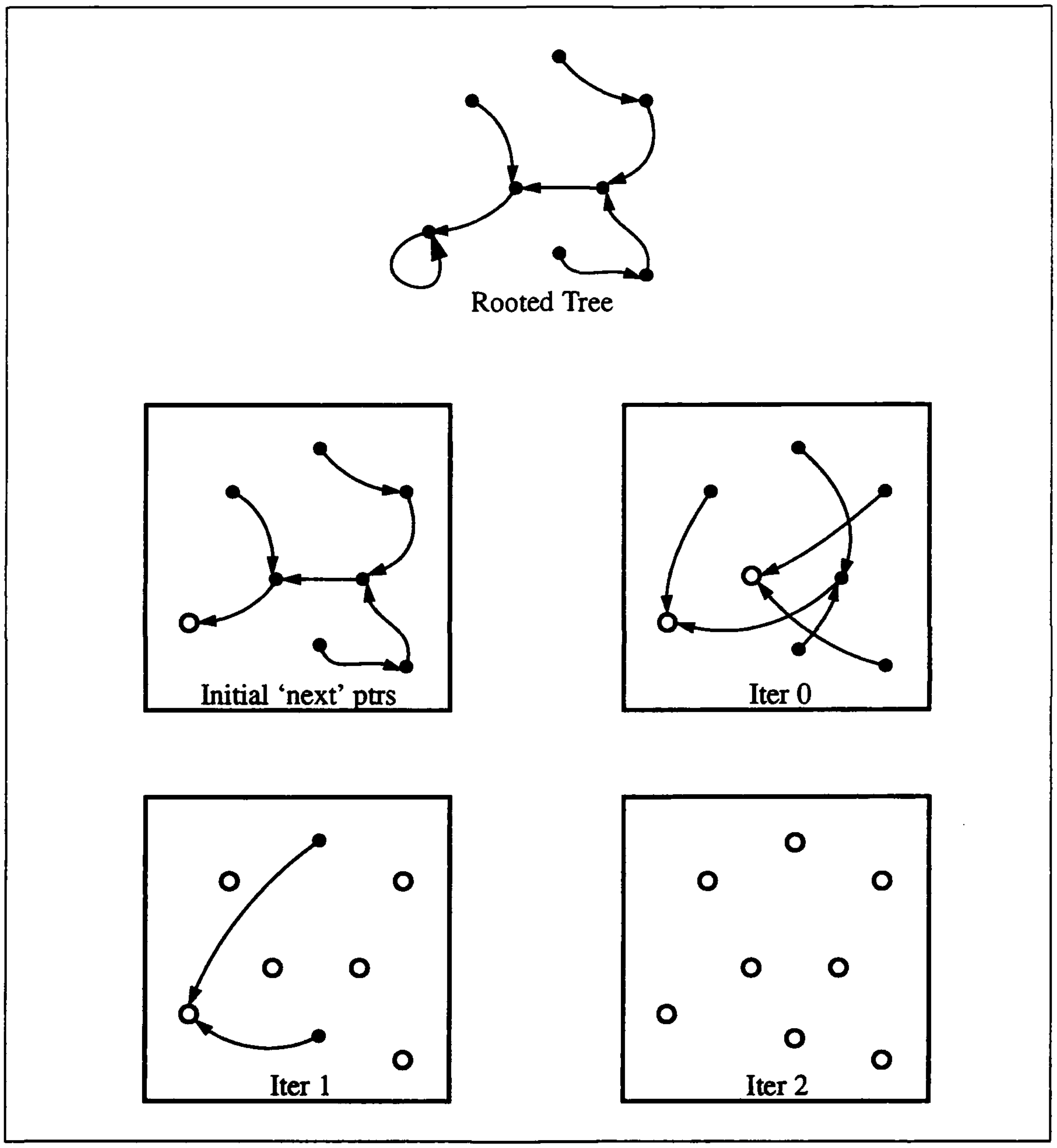


Figure 4.3: Using pointer doubling to distribute a value through a rooted tree. The root of the tree does not have a next pointer. Each doubling step transmits the root value to vertices twice as far away as the proceeding step. After $\lg n$ steps, the value is distributed to all vertices in the tree.

```
v.next = (v.ptr == v.#) ? null : v;           //next describes a list
M = match(v.next, v.#);                       //iter 1: S=O(lg n), but mapping
                                              //is EX and next is Unique
v.next = move-arb(M, v.next);                 //next is again Unique
v.val ?= move-arb(M, v.val);

M = match(v.next, v.#);                       //iter 2: S=O(1), EX mapping
v.next = move-arb(M, v.next);                 //next is again Unique
v.val ?= move-arb(M, v.val);
 :

 :
M = match(v.next, v.#);                       //iter lgN: S=O(1), EX mapping
v.next = move-arb(M, v.next);                 //next is again Unique
v.val ?= move-arb(M, v.val);
double* = v.val;
```

Figure 4.4: Pointer doubling unrolled when the pointers describe a list. After the first full **match** the deduction rules ensure that all other statements are $S = O(1)$ step complexity.

Now, in the following **move-arb** statement, the next field (which is Unique) is moved through an EXclusive mapping. The result will be Unique as well because of the rule for **move-arb** with an EXclusive mapping.

In all successive iterations, the **match** will involve an Index and a Unique vector for an $S = O(1)$ step complexity **match** that produces an EXclusive mapping. Furthermore, all successive next fields will be marked with the Unique property. So, the remaining $O(\lg n)$ iterations will each proceed with $S = O(1)$ step complexity and $W = O(n)$ work complexity. The result is that the same doubling procedure, when run on a linked list automatically has an $S = O(\lg n)$ step complexity with a $W = O(n \lg n)$ work complexity — a savings of a factor of $O(\lg n)$ steps!

In fact, this is the algorithm of Wyllie [Wyl79] that guarantees Exclusive-Read access for list contraction. The general purpose **double*** algorithm can be used whether the graph is a list or not. When it is, the inference rules ensure that the necessary optimizations are performed to achieve the complexity bounds that are possible in the EREW model.

4.6 Pragma Statements

There is a long history of the use of **pragma** statements or comments for vectorizing compilers. Generally, a vectorizable loop is recognized by matching a code fragment to some sort of template. The compiler attempts to use rules of inference to guide its matching process, but sometimes recognizing a loop as vectorizable requires directives from the programmer. This situation arises because of the generality of the operators that may be combined in a vectorized loop. By giving the programmer the ability to supply extra information to the compiler both the programmer and compiler benefit. The programmer is able to more directly communicate his intent to the compiler, and the compiler is able to apply optimizations that may have not been safe without the hints of the programmer.

The use of **match** is similar. While many common optimizations are recognized by inference rules, some more aggressive optimizations depend on characteristics of the key vector that are determined by the algorithm. A **pragma** statement allows knowledge of these characteristics to become part of the parallel program.

The preceding section showed an example in which properties were deduced automatically by the **match** operator itself. These properties can be given to the implementation directly through a **pragma** statement. To mark a field with a property, the field name and a property are given. For instance, in the pointer-doubling example of a linked list, the next field is known to be Unique. This information can be encoded in a **pragma** statement.

```
pragma(v.next, Unique);
```

Information about the creation of a mapping with **match** is given by preceding the **match** statement with **pragma** directive. Using the linked list example again, all of the **match** statements were known to be EXclusive.

```
pragma(EX);
M = match(v.next, v#);
```

Together, these two examples given here supply redundant information. It is not often the case that a field property **pragma** gives the same information as a mapping type **pragma**. When this is the case, the programmer should supply only one or the other, choosing the method that serves to best document the intent of the programmer.

Until now, the only hints mentioned for **match** operations have been the resulting mapping types. This mechanism may also be used to supply other hints, such as implementation strategies for different architectures. While the implementation is free to ignore these hints, they are often used to employ optimizations. A later section shows that a general **match** is usually implemented using either sorting or hashing to assign the groups of keys a canonical index. With a simple directive of the form **pragma**(Hash) or **pragma**(Sort) the programmer can explore the effects of these implementation strategies on different algorithms with only a minor change to the code. Notice that these hints can in no way change the *functionality* of a program implemented with **match** and **move**.

4.7 Library Functions

The only operation that produces a mapping is **match**. However, the creation of some mappings is so common that special functions are defined to produce them. Library functions are usually

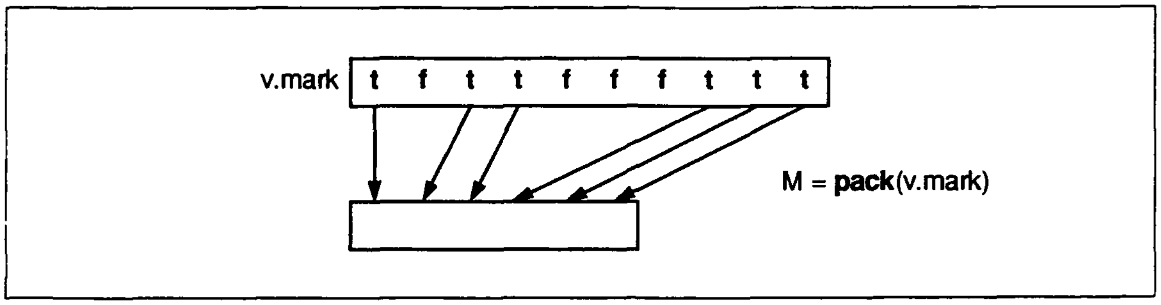


Figure 4.5: **pack** creates a mapping from the **true** sites to a new vector. The vector set of the destination is a new set that is created by the **pack** function.

implemented at a low level for higher performance, but the high level implementations shown here using the basic **match** and **move** primitives obtain the same asymptotic complexity measures as a low level implementation when the optimizations are taken into account.

4.7.1 pack

One library function builds a mapping from a Boolean field to a new vector that maps only the **true** sites to the new sites. This function is called **pack** and creates a mapping that packs the true sites into a new, smaller vector set. Figure 4.5 illustrates the effect of such a mapping. The code that implements **pack** follows.

```
pack(v.mark) {
    s.key = [true];
    M1 = match(s.key, v.mark);                        //match from Boolean to single site
    new.<junk,site> = move-join(M1, s.key, v.#);      //move indices of true sites
    M2 = match(new.site, v.#);                        //only true sites
    pack = M2;
}
```

This function is especially interesting because it has an $S = O(1)$ step complexity measure. The first **match** on Boolean fields produces a CW mapping and requires only $S = O(1)$ steps. The **move-join** moves a Unique source field (the index vector of v) through a CW mapping, producing a Unique result, new.site. The second **match** is on a Unique, Enumerable vector to an index, so it is $S = O(1)$ again. The result is that a **pack** mapping is created in $S = O(1)$ steps. This measure accurately reflects the complexity of packing in the vector model.

4.7.2 append

Two vectors can be concatenated with the **append** function. Rather than produce a mapping, this function returns a field in a new vector set. In the regular vector model, this function is implemented with some address arithmetic and two **send** operations. The same operation is defined using using **match** and **move** in a similar manner.

```
append(v1.fd, f2.fd) {
    l1 = length(v1);
    l2 = length(v2);
    new = new(l1+l2);
    new.fd = move-arb(match(new.#, f1.#), f1.fd);
    new.fd ?= move-arb(match(new.#, (f2.# + l1)), f2.fd);
    append = new.fd;
}
```

In the second **match** statement, the source is the sum of an index field and a constant: without too much difficulty a simple optimizing compiler could determine that the result is again a Unique field. Thus, the two **match** operations involve index fields and Unique fields so they have an $S = O(1)$ step complexity. Once again, this measure accurately reflects the complexity of this operation in the simpler vector model.

4.7.3 collapse

Another common function that creates a mapping is called **collapse**. This function accepts a single field of keys and returns a mapping to a new vector set. For each distinct key in the field, a destination site is created in the mapping, so that all sites with the same key map to the same site in the destination vector. The length of the destination vector is exactly the number of distinct keys in the key vector. An illustration of a mapping produced by **collapse** appears in Figure 4.6. This operation forms the basis for a simple histogramming operation. By applying the mapping created by **collapse** to a vector of 1's with **move-add**, the destination vector will contain a count of the number of each key in the source. The implementation of this function is essentially the same as that presented by Sabot [Sab88b].

```
collapse(v.key) {
    M1 = match(v.key, v.key);
    v.arb = move-arb(M1, v.#);
    v.mark = (v.# == v.arb);
    new.key = move-arb(pack(v.mark), v.key);
    M2 = match(new.key, v.key);
    collapse = M2;
}
```

The first **match** creates a mapping from sites with the same key back to themselves. Then, for each group of equal keys, the next step moves an arbitrary index of one back to all matching

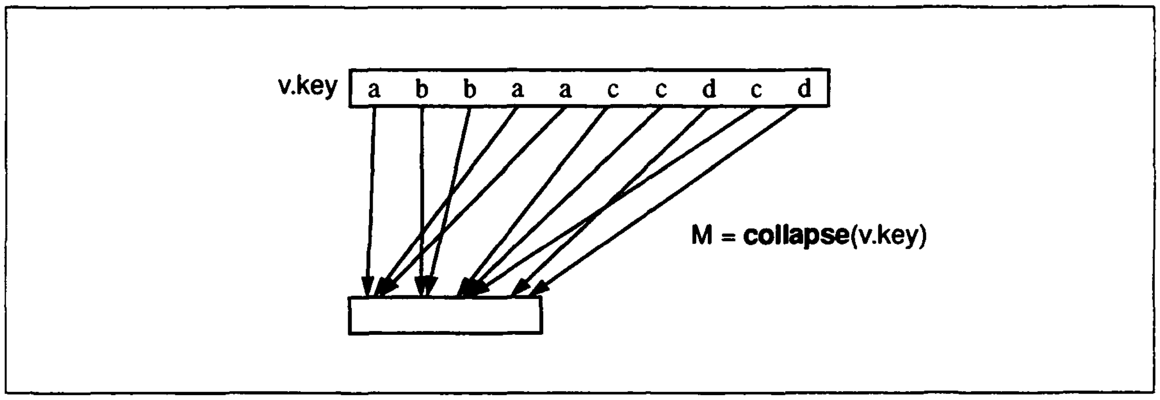


Figure 4.6: **collapse** gathers equal keys together. This function is generally used to remove duplicate keys or as a precursor to a histogramming operation using a combining **move**.

keys. The key that gets its index back is marked as special; only one of each group can be special. The **pack** and **move-arb** then move the indices of these special sites to a smaller vector that has one site for each unique key of the original. The final step creates a mapping from the original keys to the new smaller vector.

There are three mappings created in this function. Two are created directly using **match** (M1 and M2) and one is the result of **pack**. The **pack** operation requires only $S = O(1)$ steps, but the step complexities of the other two **match** operations are dependent on properties of the key vector and determine the step complexity of this function. If the keys are Boolean, then **collapse** can be implemented in $S = O(1)$ steps. If the keys are Enumerable, then the step complexity is $S = O(\lg R)$, otherwise, the step complexity is $S = O(\lg n)$. In all cases, the work complexity is related to the number of keys and the step complexity, $W = O(Sn)$

Part II

Algorithms and Data Structures

Algorithms and Data Structures

Part II of this dissertation shows examples of many algorithms that are easily described using **match** and **move**. The tools of the preceding chapters are used to evaluate each of the examples for their algorithmic complexity measures. The measures derived show that writing these algorithms with **match** and **move** obtains the same efficiency measures as the best PRAM algorithms.

The chapters of this part divide the algorithms into groups by subject. Chapter 5 is a gentle introduction to fundamental algorithms on sets and sparse vectors. With these techniques, a wide variety of graph algorithms are presented in Chapter 6. The following chapter concludes this part with a presentation of sparse matrix techniques, leading to a parallel Gaussian Elimination algorithm for solving linear systems as well as a large number of path problems on graphs.

Chapter 5

Basic Techniques

Algorithms described using **match** and **move** are based on simple collections of data objects that use **match** to describe relationships between those objects. In this representation, an object is represented by a tuple and a collection of like objects is stored in a vector of tuples. For each object, some of the fields in the tuple are keys by which the object is referenced, while others are data values associated with the object. The designation of a field as a key or a data value occurs solely in the context of a field's use. In some situations, a particular field may be used as a key for a match operation and then later treated as a data value. For many algorithms, however, the role of key fields and data fields is more clearly defined.

A flexible framework for defining data structures is provided through the use of keys that permits an associative model of computing similar to that proposed by Potter [Pot88]. Keys can provide the functionality of pointers, or can serve to label objects that belong to the same row, column or other type of class. This chapter shows how this very general mechanism can be used to implement some important fundamental data types. Using the notion of keys and tuples, a wide variety of algorithms on sets, vectors, graphs, trees and matrices can be implemented. The expressive power and versatility of the **match** primitives allow most of these algorithms to be described in just a few lines of pseudocode.

Figure 5.1 provides an orderly classification of the algorithms we will present. They are grouped according to function. The precedence relation shown by the arrows indicates that an algorithm at the head of an arrow builds from a technique at the arrow's tail. The order of presentation attempts to build from the simplest component techniques to the most complicated of the algorithms. Because the later algorithms build on the earlier ones, the chapters of this part should be read in sequence.

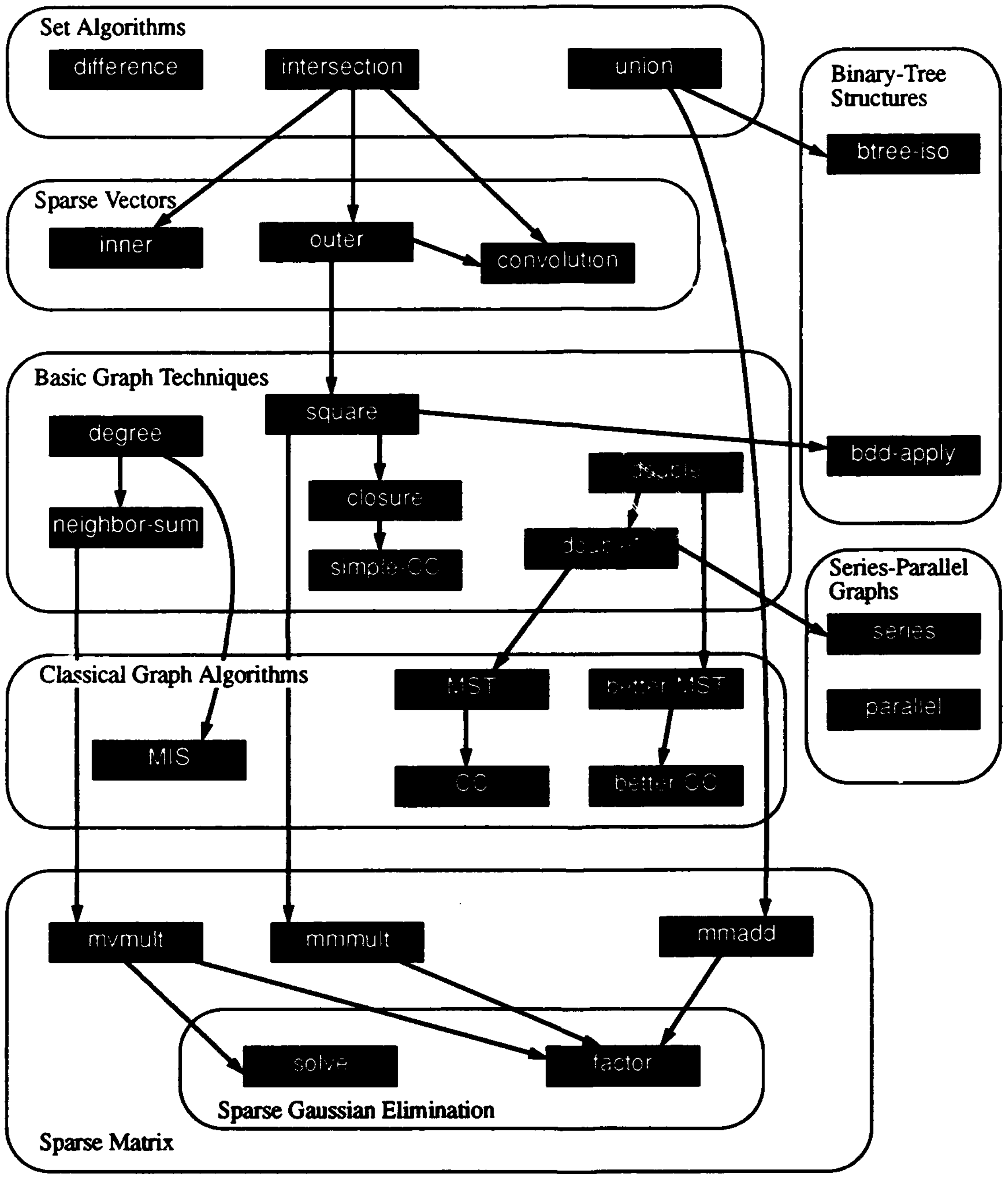


Figure 5.1: A taxonomy of the algorithms presented in Part II. Where the techniques introduced in one algorithm influence the implementation of another, a precedence relation is shown with an arrow.

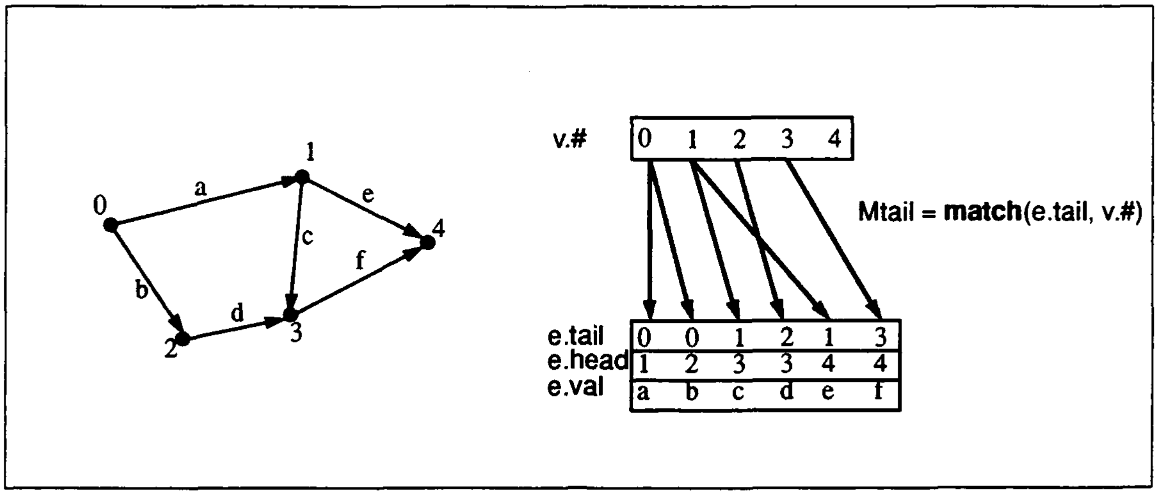


Figure 5.2: Sparse graph representation. The id of each vertex is its index, so **match** operations involving vertices result in simple, direct mappings.

5.1 Data Structures

A directed graph $G = \langle V, E \rangle$ is described by a set of vertices V and edges E. Generally, for a graph G with $n = |V|$ and $m = |E|$, the vertices are assigned integer identifiers (ids) in the range $[0..(n-1)]$. Each edge $(i, j) \in E$ is directed from its tail, i, to its head j. If a graph has duplicate edges between any two vertices then it is called a multigraph, otherwise it is simple. In an undirected graph, each direction of an edge is stored explicitly. This is called the "bidirectional" representation of an undirected graph.

The set of vertices, V, is represented as a vector of length n. Because the id of each vertex is its index in the vector, no special key field need be associated with this vector. Additional fields are used when there are other labels or values associated with the vertices.

The set of edges, E, is represented by a vector with tail and head fields, while additional fields store data values for each edge. An example is shown in Figure 5.2. Using **match**, mappings are created to describe how vertices are related to edges and how edges are related to vertices. In the example, the mapping Mtail relates each vertex to the edges for which it is a tail. This mapping could be used to move data from vertices to the sites of each of its outgoing edges.

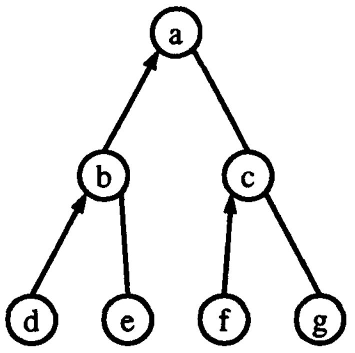
A sparse matrix is described by a collection of its non-zero entries. Each is identified by column and row keys and has an associated value. Figure 5.3 shows such a sparse matrix. Even though the storage of the row and column keys require more memory per element than a typical dense arrangement, this scheme is more efficient for many sparse matrices of interest. Using this representation, mappings can be created that distribute values along rows or columns, or that sum (reduce) elements along either of these dimensions simply by **matching** on the appropriate keys. Matrices of higher dimension are represented by adding more keys.

A regular binary tree is a collection of nodes, each of which has two children called its *left*

$$\begin{bmatrix} a & 0 & e \\ 0 & c & 0 \\ b & d & 0 \end{bmatrix}$$

m.col	0	0	1	1	2
m.row	0	2	1	2	0
m.val	*a*	*b*	*c*	*d*	*e*

Figure 5.3: Sparse matrix representation. Only non-zero values are stored; each is identified with its row and column.



t.id	a	b	c	d	e	f	g
t.left	b	d	f	-	-	-	-
t.right	c	e	g	-	-	-	-

Mleft = **match**(t.id, t.left)

Figure 5.4: Example binary tree. A parent-child relationship is indicated where the left or right field of a node references the id of another. The **match** indicated creates a mapping from each left child to its parent.

and *right* child. Using tuples, it is also easy to represent a binary tree. Each node of the tree is given a unique label called id. Key fields called left and right reference the id fields of other nodes in the tree as shown in Figure 5.4. If a node is a leaf, then its left and right children have the **null** key. Using the keys, mappings that describe parent-child relationships are created as needed. For example, a mapping that moves data from each left child to its parent is created as shown in the figure. Such an operation is the parallel analogue of pointer dereferencing.

5.2 Set Operations

A set of n elements is represented by a simple vector of n values, in which there may be no duplicates. In this section, no assumptions are made about the type of the values but there must be an equality function for comparing elements of the two sets. The basic set operations

```
union(s1.key, s2.key) {
    temp.key = append(s1.key, s2.key);          //append
    M = collapse(temp.key);                     //remove duplicates
    union = move-arb(M, temp.key);
}

intersection(s1.key, s2.key) {
    M = match(s2.key, s1.key);                  //pairs match equal elements
    temp.<s1,s2> = move-join(M, s2.key, s1.key); //produce a new site for each pair
    intersection = temp.s1;
}

difference(s1.key, s2.key) {
    M1 = match(s1.key, s2.key);
    s1.mark = true;                             //distribute to all sites
    s1.mark ?= move-arb(M1, false);             //mark matching elements
    M2 = pack(s1.mark);                         //keep only those marked
    difference = move-arb(M2, s1.key);
}
```

Algorithm 5.1: Basic set functions. The elements of the sets are also the keys used in all of the **match** operations.

(**union**, **intersection** and **difference**) are easily described in just a few lines of code as shown in Algorithm 5.1.

The union of two sets is made by first appending the vectors of the two sets and then removing any duplicates that occur. The library function **collapse** creates a mapping that sends equal keys to the same site in a new vector. By using **collapse** and **move-arb**, duplicate entries are removed from the appended vector.

The intersection of two sets is most easily implemented by taking advantage of the special properties of the **move-join** operator. Because each of the argument sets may have no duplicates, the mapping created by matching from one set to the other is EXclusive (with some elements not mapping to anything). When **move-join** is applied to this type of mapping, exactly one new site is created for each destination site that matches a source site. Figure 5.5 illustrates the simplicity with which **move-join** allows this operation to be implemented.

To describe set difference, a more straightforward application of **match** and **move-arb** is used to mark only those elements with a mate in the other vector. The elements not marked are then packed into a new vector. This approach is illustrated in Figure 5.6. Of course, set intersection could have been implemented this way by simply inverting the sense of the mark field, but the use of **move-join** there exposed some of the uses of that operator.

While the implementation of these operations is nearly trivial, they are important kernels for more complex algorithms. Often, the set elements are the keys of objects and the desired

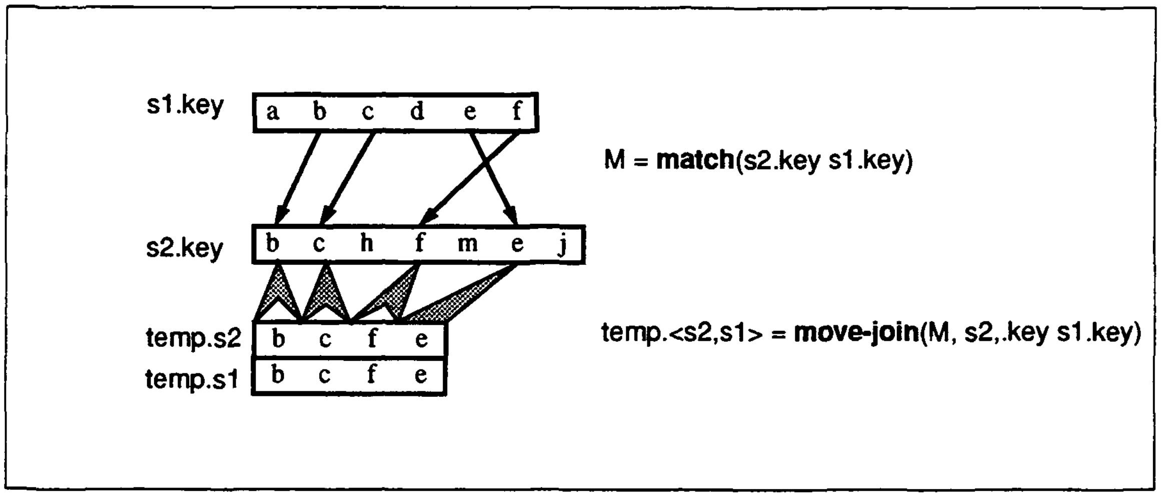


Figure 5.5: Set Intersection. The **move-join** operation ensures that new sites are created only where set elements match.

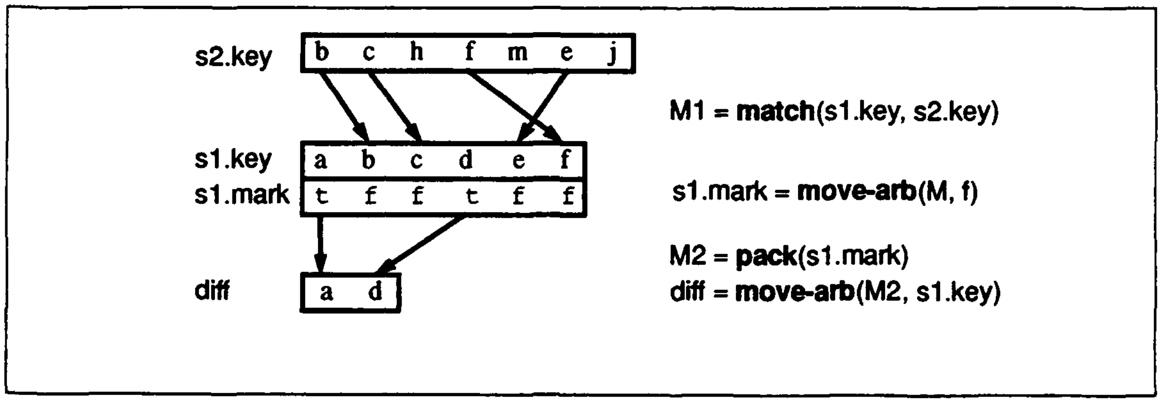


Figure 5.6: Set Difference. Marks are used to indicate elements in one set that are not in the other. Unmarked elements are then removed by using a packing step.

```
inner(a.<row,val>, b.<row,val>) {
    M = match(a.row, b.row);
    temp.<a,b> = move-join(M, a.val, b.val);
    temp.val = temp.av × temp.bv;
    inner = reduce-add(temp.val);
}

outer(a.<row,val>, b.<row,val>) {
    a.const = true;
    b.const = true;
    M = match(a.const, b.const);                            //create all cross pairs
    temp.<col,a,row,b> = move-join(M, a.<row,val>, b.<row,val>);
    temp.val = temp.av × temp.bv;
    outer = temp.<col,row,val>;
}

convolution(a.<row,val>, b.<row,val>) {
    temp.<col,row,val> = outer(a.<row,val>, b.<row,val>);  //all cross products
    temp.key = temp.row + temp.col;
    M = collapse(temp.key);                                 //gather those with same key
    new.val = move-add(M, temp.val);
    new.<row,col> = move-arb(M, temp.<row,col>);
    convolution = new.<row,col,val>;
}
```

Algorithm 5.2: Sparse vector functions. Both **inner** and **outer** make use of **move-join**, but in radically different ways.

operation is to make a set of objects whose keys have some property. By using **match** on the keys, the mappings created can be used to move the other fields of the objects to their new sites.

5.2.1 Complexity

Because each of these operations use one full **match** and one **move**, the step and work complexities of each is $S = O(\lg n)$ and $W = O(n \lg n)$, where n is the total number of elements in the two sets.

5.3 Sparse Vector Algorithms

Each element of a sparse vector is a tuple with two fields giving its row key, row, and an associated value, val. With this simple representation it is possible to easily describe sparse analogues of some typical dense vector operations. The operations considered here are **inner** product, **outer** product, and **convolution**. Their implementations using **match** and **move** are shown in Algorithm 5.2.

The inner product of two vectors sums the products created between elements with equal row

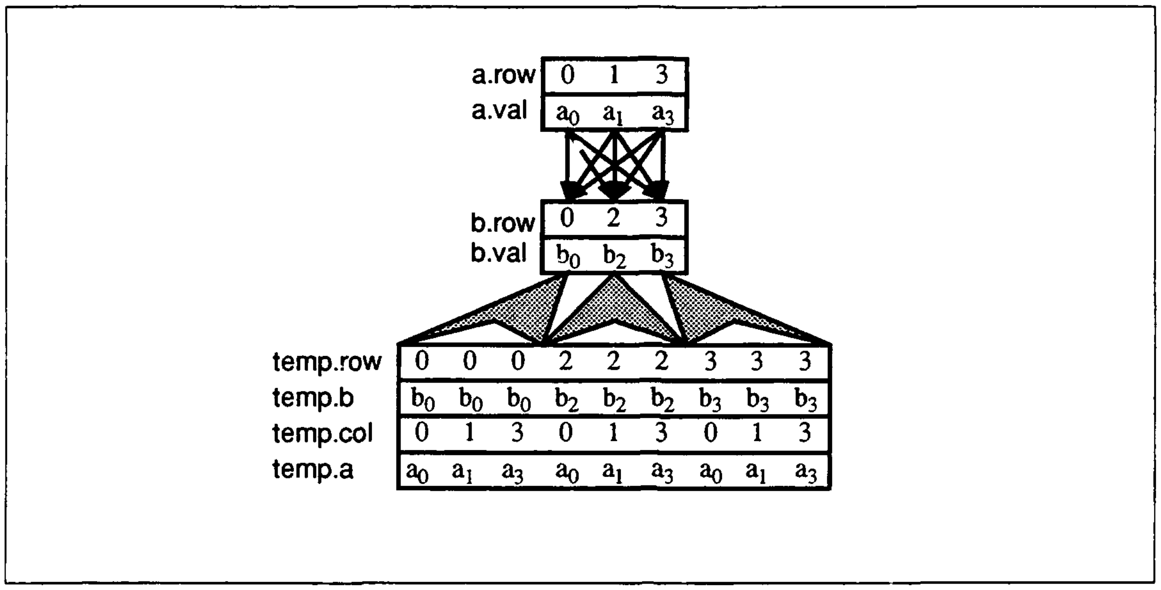


Figure 5.7: Outer Product of Two Sparse Vectors. Because the mapping is a many-to-many, application of **move-join** creates all pairs of the argument vectors.

indices to produce a scalar result. In general, for two sparse vectors a and b, the inner product, I, is computed by the following formula:

$$I = \sum_i \{a_i b_i \mid a_i \in A, b_i \in B\}.$$

When the vectors are sparse, a product need not be computed whenever there is not a corresponding element in both of the vectors. By matching on the row key and using **move-join**, the sites needed for the necessary products are created in just two steps. The reduction of the products produces the scalar result. By producing sites where only the row keys match, the use of **move-join** in this example is essentially the same as its instance in **intersection**.

The outer product of two dense vectors is the matrix that contains entries for each possible pair of entries of the argument vectors. For two sparse vectors a and b of size N and M, respectively, their outer product, O, is the sparse matrix whose values are described by the set

$$O = \{a_i b_j \mid a_i \in A, b_j \in B\}$$

To create the cross products between all elements that are present, a mapping is created in which all of the elements in the first vector match all of the elements in the second. Then, using **move-join**, sites are created for all possible pairings of the two and the products are computed. The last step simply moves the appropriate values into the tuple slots. A diagram of the data movement this function performs is shown in Figure 5.7. While not showing the actual final vectors, it shows all of the values moved to the necessary result sites.

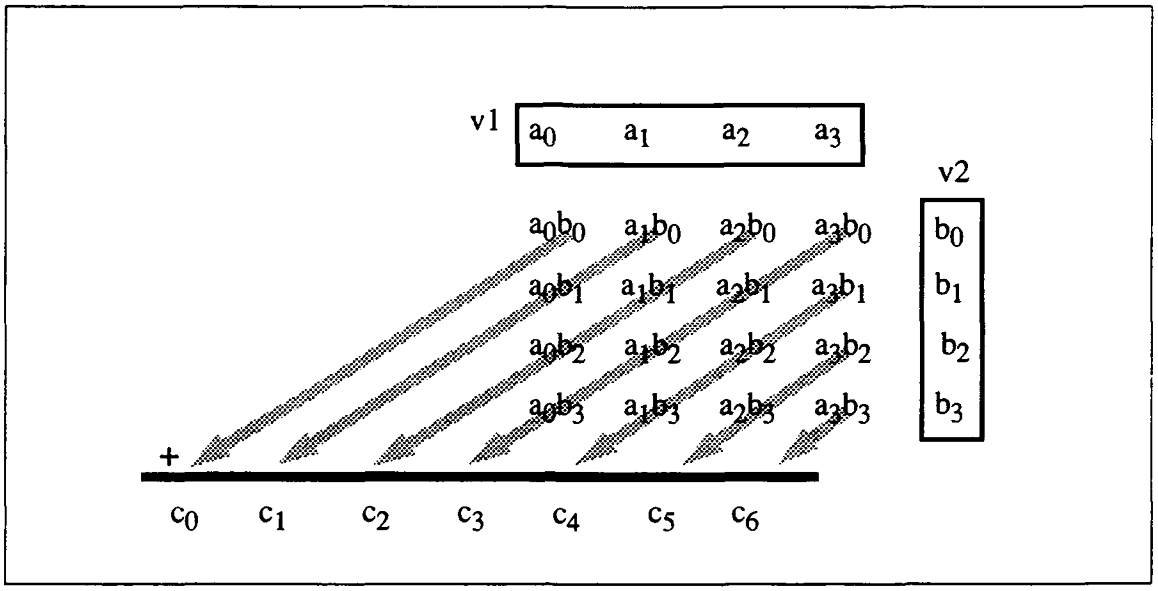

v1
a_0 a_1 a_2 a_3
v2
b_0 b_1 b_2 b_3
a_0b_0 a_1b_0 a_2b_0 a_3b_0
a_0b_1 a_1b_1 a_2b_1 a_3b_1
a_0b_2 a_1b_2 a_2b_2 a_3b_2
a_0b_3 a_1b_3 a_2b_3 a_3b_3
+
c_0 c_1 c_2 c_3 c_4 c_5 c_6

Figure 5.8: Convolution of Two Dense Vectors. After creating all necessary cross products, the values of the convolution vector are the sums along the diagonals indicated.

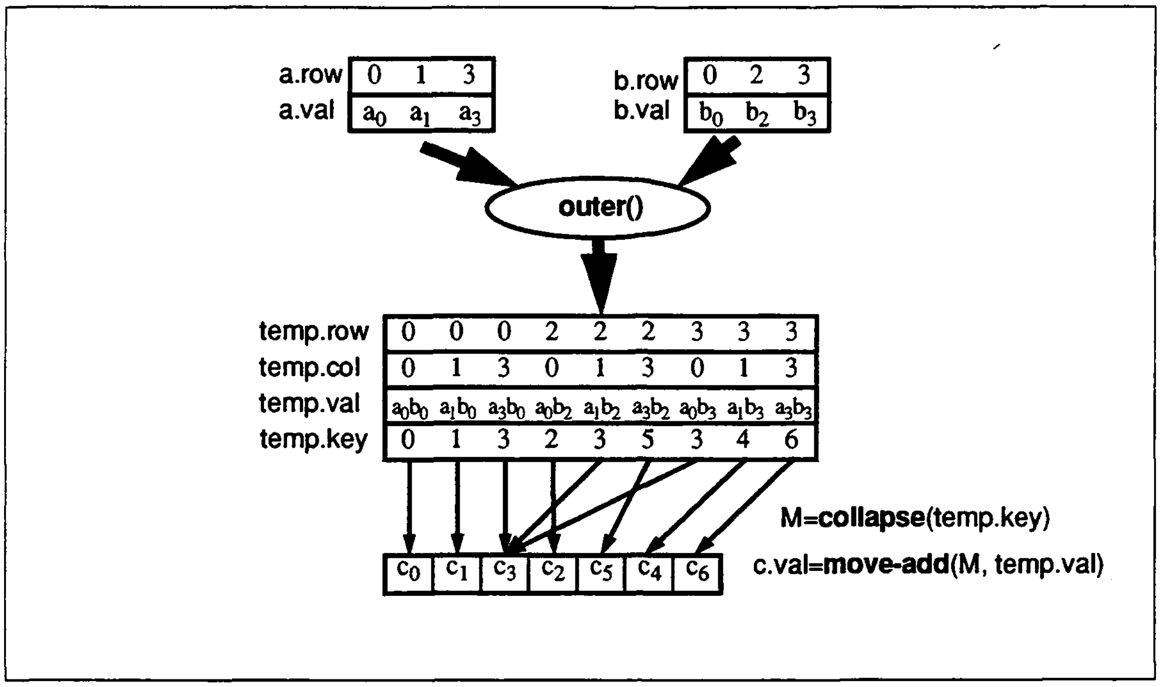
Convolution is an important operation in image and signal processing. If the two vectors represent the coefficients of two polynomials, then convolution of the vectors also yields the coefficients of the product of the polynomials. The convolution of two dense vectors of length N is a vector of length $(2N - 1)$ whose elements are described by the following equation:

$$C_k = \sum\{a_i b_j, i + j = k\}.$$

For dense vectors, this operation has an appealing graphical representation. After creating the cross products for each pair of argument elements, result elements whose source indices sum to the same value are added together. Laid out as an array, the elements of the convolution are sums of the diagonals of the array as shown in Figure 5.8. When the vectors are sparse, the total number of result elements is dependent on the manner in which the subscripts of elements of the two vectors match one another. However, the total number of partial products computed is still the product of the lengths of the vectors. The set of result elements for sparse vector convolution is described by:

$$C_k = \sum\{a_i b_j \mid a_i \in A, b_j \in B, i + j = k\}.$$

Using **outer**, all of the necessary cross products are easily created. Then, a key is calculated for each element based on the sum of its row and column. Using this computed key with **collapse**, elements along the diagonals are combined with **move-add** to produce the final result elements. The diagram in Figure 5.9 illustrates the products resulting from **outer** and the manner in which they are added together.



a.row 0 1 3
a.val a0 a1 a3
b.row 0 2 3
b.val b0 b2 b3
outer()
temp.row 0 0 0 2 2 2 3 3 3
temp.col 0 1 3 0 1 3 0 1 3
temp.val a0b0 a1b0 a3b0 a0b2 a1b2 a3b2 a0b3 a1b3 a3b3
temp.key 0 1 3 2 3 5 3 4 6
M=collapse(temp.key)
c.val=move-add(M, temp.val)
c0 c1 c3 c2 c5 c4 c6

Figure 5.9: Convolution of Two Sparse Vectors. All cross products are created using **outer**. Elements that would be on the same diagonal are indicated where the sums of the subscripts are equal.

5.3.1 Complexity

Even though each of the sparse vector algorithms consist of only a few lines of code, their complexity measures demonstrate the variety with which **match** and **move** can be combined to create algorithms. In this analysis, the two vectors are of length n_1 and n_2, and the total number of elements is $n = n_1 + n_2$. Function **inner** performs a full **match** on the elements. Hence, it has $S = O(\lg n)$ step complexity and $W = O(n \lg n)$ work complexity.

Function **outer** is interesting because the only **match** performed is on a vector of Booleans. This is a special example of a fixed-range **match**, so its step complexity is $S = O(1)$. Its work complexity is related to the total number of elements produced, $W = O(n_1 n_2)$. The **convolution** function makes use of **outer**, but because the **match** is on integer keys, its step complexity is $S = O(\lg(n_1 n_2))$. Similarly, the work complexity is determined by the total number of new elements produced. These measures are displayed in Table 5.1.

Table 5.1: Complexities of the Sparse Vector Operations for two vectors of length n_1 and n_2. The total number of elements is $n = n_1 + n_2$.

Function	Step	Work
inner	$O(\lg n)$	$O(n \lg n)$
outer	$O(1)$	$O(n_1 n_2)$
convolution	$O(\lg(n_1 n_2))$	$O((n_1 n_2) \lg(n_1 n_2))$

Chapter 6

Graph Algorithms

This chapter first presents some very simple operations on graphs and then progresses to more complicated algorithms. Following standard graph terminology [Har69], a directed graph G is represented by a set of edges E, and a set of vertices V. Each directed edge, (i, j), from vertex i to vertex j has a tail i, a head j, and an associated value. The set of edges incident to a vertex are called its "in-edges" while the set of edges leaving a vertex are called its "out-edges." A path in G is a sequence of vertices $< i, j, \cdots, k >$ such that there is an edge (i, j) and there is a path from j to k. The length of a path is the number of edges it contains. A simple path contains no cycles. A path is assumed to be simple unless otherwise noted.

In this simplified representation, each edge is a tuple with fields <tail,head,val>, such that the tail and head reference vertices through their vector index. Each vertex will have only a single field associated with it: its value. Figure 5.2 showed an example of such a directed graph. In the algorithms presented here, undirected graphs are expanded into bidirectional graphs by duplicating each edge for the two directions implied by the undirected edge.

It is often the case that a single mapping is used repeatedly in an algorithm. In fact, it is desirable to amortize the cost of a **match** over multiple uses of **move**. Almost all graph algorithms make use of a small set of simple mappings between edges and vertices. These four mappings are called Mtail, Mhead, Min, and Mout, and describe the relations of vertices to edges, and edges to vertices.

```
Mtail  = match(e.tail, v.#);
Mhead = match(e.head, v.#);
Min    = match(v.#, e.head);
Mout   = match(v.#, e.tail);
```

For each vertex, mappings Mtail and Mhead move values to those edges for which it is a head or tail, respectively. Similarly, mappings Min and Mout move values from the in-edges or out-edges of a vertex, combining them at the destination vertex. Because the labels of the

```
degree (Min) {
    degree = move-add(Min, 1);                   //count incoming edges
}

neighbor-sum (Mtail, Min, v.val) {
    e.copy = move-arb(Mtail, v.val);             //copy vertex to out-edges
    neighbor-sum = move-add(Min, e.copy);        //sum on in-edges
}
```

Algorithm 6.1: The degrees of all vertices and the sums of all neighbors. Both of these functions use the mapping Min to sum the values on the in-edges of each vertex. To calculate the degree, this is a constant. To sum the neighbors, each neighbor sends its value to its out-edges, which are in-edges of its neighbors.

vertices are known to be an index field, which is Unique, these mappings are simple concurrent read or concurrent write operations. The implementation of **match** can use this information if the underlying machine supports these operations directly.

In general though, for a source vector of length s and a destination of length d, an application of **match** requires $S = O(\lg(s + d))$ steps in the vector model. Because a graph with n vertices may have no more than $m \leq n^2$ edges, this reduces to

$$S = O(\lg n + \lg n^2) = O(\lg n).$$

For algorithms on graphs, the step complexity of creating these common mappings is related to the total number of vertices, while the work complexity is related to the total number of elements in the graph. For a graph of e edges, the work complexity is generally $W = O(e)$.

6.1 Fundamental Graph Algorithms

6.1.1 The Degree of a Vertex

The degree of a vertex is the number of its "in-edges." This number may be computed for each vertex with the function **degree** shown in Algorithm 6.1. This function takes only a mapping as an argument. Because scalars are promoted to a vector of constants where needed, and **move-add** expects a vector of the length of the source, the scalar 1 is distributed across the edges. The result of the **move-add** is a vector describing the degree of each vertex. Notice that the same function can compute the number of "out-edges" of a vertex with the mapping Mout.

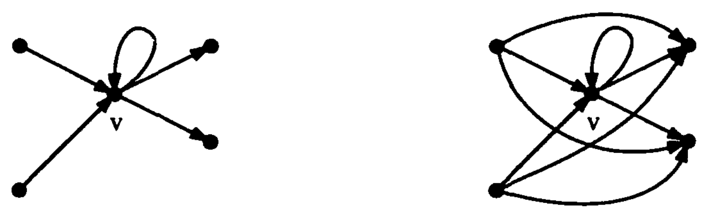


Figure 6.1: Squaring the edges incident to vertex v in a directed graph. A new edge is created wherever the head of an edge matches the tail of another.

6.1.2 Neighbor Summing

The sums of the neighbors of each vertex are almost as easily computed. This function is shown in Algorithm 6.1. Using the mapping Mtail, the **move-arb** distributes to each edge the value of its tail vertex. This replicates the values of the vertices for each of its outgoing edges. Then, a simple **move-add** with the mapping Min sums the values of the neighbors of each vertex. This procedure can be slightly modified to scale the vertex values by the edge values before they are summed.

6.1.3 Graph Squaring

An adjacency graph indicates only connectivity between vertices; as such, the edges have no associated values. Because vertices must be adjacent to themselves, all vertices also have an edge directed from and to themselves. Such a directed graph is often used to represent a reflexive binary relation.

Conventionally, the "square" of an adjacency graph, G^2, is a new graph with edges wherever there is a path of length 2 in G. Similarly, G^2 may be squared again, yielding G^4, to find paths in G whose lengths are 4. If self-edges are included in the graph, the square G^2 indicates paths in G of length *no greater* than 2 in G. The fundamental operation in graph squaring is the creation of a new edge wherever the head of one edge meets the tail of another edge. Figure 6.1 demonstrates the squaring operation for the edges incident to vertex v. The squared edges are those that result from all combinations of the in-edges and out-edges at vertex v. Because the self-directed edge at v is both an incoming and outgoing edge, it combines with all of the incoming and outgoing edges. Notice that wherever a self-directed edge combines with one of these edges, the same edge is created again, and that where it combines with itself, it produces itself as a result. In this sense, the self edges are identity elements for edge multiplication.

The key insight to implementing the graph squaring operation is to recognize that new edges are created wherever tails and heads "match." The trivial code for the graph squaring function

```
square (e.<tail,head>) {
    M1 = match(e.head, e.tail);                         //new edge where head meets tail
    temp.<tail,head> = move-join(M1, e.tail, e.head);
    M2 = collapse(temp.<tail,head>);                    //gather up parallel edges
    square.<tail,head> = move-arb(M2, temp.<tail,head>);
}
```

Algorithm 6.2: Squaring an adjacency graph. A new edge is created where the head of one matches the tail of another. One application of **move-join** creates sites for all of these new edges. Duplicates are removed with **collapse.**

is in Algorithm 6.2. After matching edges by head and tail, the **move-join** function is used to create sites for all of the cross products and the heads and tails of the new edges are taken from the appropriate sources. Because duplicate edges may be introduced, they are removed with the **collapse** operation near the end of the function.

When analyzing the complexity of this function, it is difficult to put a tight bound on the number of edges created by the **move-join** operation. Because no vertex may have any more than n edges, no vertex may be the pivot about which more than n^2 new edges are created. Thus, for n vertices, a total of n^3 itermediate edges could be created by **move-join** in the worst case. For very sparse graphs though, this is almost never the case. But because $O(\lg n^3) = O(\lg n)$, the step complexity of this function in $S = O(\lg n)$, while the work complexity is $W = O(n^3 \lg n)$. If the total number of intermediate edges created is m, then the work complexity is also bounded by $W = O(m \lg n)$.

6.1.4 Reflexive and Transitive Closure

Function **square** may be used to solve two problems on adjacency graphs. The first is reflexive and transitive closure [DNS81]. Given a graph G, this problem asks for a new graph H such that if there is a directed path $< i, j >$ in G, then there is an edge (i, j) in H. In a graph G of N vertices, no simple path may be of length greater than N. Because the **square** function computes a graph G^2 that has an edge for each path in G no greater than 2, **square** may be used to compute successively longer paths in G. After i applications of **square**, the graph G^{2^i} represents all paths whose length is no greater than 2^i. With $\lceil \lg N \rceil$ applications of **square**, all paths of length N will be represented explicitly. The algorithm, called **closure**, is shown in Figure 6.3.

Because of the complexity measures of **square**, this function has step complexity $S = O(\lg^2 N)$ and worst-case work complexity $W = O(n^3 \lg^2 n)$.

```
closure (e.<tail,head>, v.#) {
    lgN = lg(length(v.#));
    temp.<tail,head> = e.<tail,head>;
    for (i = 0; i < lgN; i++) {
        temp.<tail,head> = square(temp.<tail,head>);
    }
    closure = temp.<tail,head>;
}
```

Algorithm 6.3: Reflexive and transitive closure of a graph. Each application of **square** creates an edge for every path of length 2 in the previous graph. After $\lg n$ applications, all paths in the original graph are indicated by an edge in the result.

```
simple-CC (e.<tail,head>, v.#) {
    c.<head,tail> = closure(e.<tail,head>, v.#);
    M = match(c.head, c.tail);
    simple-CC = move-min(M, v.#);
}
```

Algorithm 6.4: A simple Connected Components algorithm. The **closure** function does most of the work. This algorithm uses **move-min** to label each vertex with the minimum numbered vertex in its component.

6.1.5 Connected Components

The connected components of an undirected graph are disjoint sets of vertices such that if there is a path from vertex i to j, then i and j are in the same connected component. The connected component to which each vertex belongs is identified by a single arbitrary vertex of its component: typically this is the minimum numbered one. An undirected graph is represented in bidirectional form by explicitly storing both the forward and backward versions of each edge. After using function **closure** on such a graph, there is an edge from each vertex to all others in its connected component. The minimum numbered vertex adjacent to each is calculated in a straightforward manner using **match** and **move-min**. This procedure is implemented in a function called **simple-CC** shown in Figure 6.4.

While the graph squaring approach to solving the transitive closure and connected components problems is intuitive and simple, it often results in dense graphs with large edge sets. It is for this reason the step and work complexities of this algorithm are $S = O(\lg^2 N)$ and $W = O(n^3 \lg^2 n)$. More efficient methods to solving these same problems are shown later.

6.1.6 Complexity

The complexities of the algorithms presented in this section are shown below in Table 6.1 for graphs of n vertices and m edges.

Table 6.1: Complexities of the basic Graph Functions. Algorithms based on graph squaring, including **closure** and **simple-CC** are not work-efficient. These algorithms are handled more efficiently in later sections.

Function	Step	Work
degree	$O(\lg n)$	$O(n + m \lg n)$
neighbor-sum	$O(\lg n)$	$O(n + m \lg n)$
square	$O(\lg n)$	$O(n^3 \lg n)$
closure	$O(\lg^2 n)$	$O(n^3 \lg^2 n)$
simple-CC	$O(\lg^2 n)$	$O(n^3 \lg^2 n)$

6.2 Maximal Independent Set

An independent set in an undirected graph is a subset of the vertices in which no pair are adjacent. Such a set is maximal if no other vertex may be added that maintains the adjacency requirement. A parallel algorithm for the maximal independent set (MIS) problem was given by Luby [Lub86], and is easily adapted to the vector framework. This algorithm is shown in Algorithm 6.5.

An undirected graph G is represented in its bidirectional form with a vector of vertices and a set of directed edges with each direction stored explicitly. Sets of vertices are identified in this algorithm by Boolean fields associated with the vertices. A **true** value in one of these fields identifies the vertex as belonging to a particular set. With this representation, the union and intersection of sets of vertices can be calculated using simple elementwise Boolean operations.

The current set of vertices accepted into the MIS is given by field I. Initially this set is empty; at the completion of the algorithm it identifies the independent set. During each iteration of the loop of the main function a set of new vertices, marked in field II, become candidates for the independent set. These vertices are then merged into I.

As the algorithm proceeds, parts of the graph are pruned away. The set Y contains those vertices that are part of the currently active graph, and only these are candidates for becoming members of the independent set in the next pass. Initially this set contains all of the vertices. As vertices are selected to be in the independent set (IS), they and their neighbors are removed from the active set. The edges induced on the active set of vertices, Y, are all that need to be considered in the next pass. The algorithm terminates when there are no more edges to be considered.

Before describing the main loop of the algorithm, a number of ancillary functions are defined. A function called **neighbors** returns a new set that contains vertices adjacent to a given set

```
neighbors(v.marks, Mtail, Min) {                    //mark neighbors of marked verts
    v.temp = move-arb(Mtail, v.marks);              //distribute to edges
    neighbors = move-or(Min, v.temp);               //OR into verts
}

degree(Min) {                                       //simple degree calculation
    degree = move-add(Min, 1);
}

induce-subgraph(v.marks, e.<tail,head>, Mtail, Mhead, Min) {
    e.mh = move-arb(Mhead, v.marks);                //mark edges with marked head
    e.mt = move-arb(Mtail, v.marks);                //... with marked tail
    e.mark = e.mh & e.mt;                           //save edges with both marked
    induce-subgraph = move-arb(pack(e.mark), e.<tail,head>);
}

select(Mtail, Mhead, Min) {                         //probabilistic selection
    v.d = degree(Min);
    v.X = coin(v.d);                                //based on degree
    e.<tX,td> = move-arb(Mtail, v.<X,d>);
    e.<hX,hd> = move-arb(Mhead, v.<X,d>);
        //deselect a tail if its head is of higher degree
    e.deselect = ((e.hX & e.tX) & (e.td ≤ e.hd)) ? true : false;
        //vertex remains selected only if not deselected by an edge
    v.X = v.X & !(move-or(Min, e.deselect));
    select = v.X;
}

MIS(v, e.<tail,head>) {
    v.I = f;                                        //in currently selected MIS?
    v.Y = t;                                        //is a currently active vertex?
    while (length(e) ≠ 0) {
        Mhead = match(e.head, v.#);                 //build standard mappings
        Mtail = match(e.tail, v.#);
        Min = match(v.#, e.head);
        v.II = v.Y & select(Mtail, Mhead, Min);     //select candidate vertices
        v.I = v.I | v.II;                           //add to MIS set
        v.Y = v.Y & !(v.II | neighbors(v.II, Mtail, Min));  //remove from active set
        e.<tail,head> = induce-subgraph(v.Y, e, Mtail, Mhead, Min);
    }
    MIS = v.I;
}
```

Algorithm 6.5: Maximal Independent Set algorithm. Vertices are successively added to the independent set with a probability inversely proportional to their degree. The algorithm terminates when no more vertices are candidates for inclusion in the independent set.

identified by a vector called marks. This function is implemented by first moving the value of the Boolean flag to the outgoing edges of each vertex so that an edge is marked if its tail is marked. Then, new marks are computed by or-ing together the values of the incoming edges of each vertex by using mapping Min. This function is similar to **neighbor-sum** showed earlier except that Boolean, rather than arithmetic, operations are used.

The set of edges induced on a set of vertices is returned by **induce-subgraph**. The induced subgraph of a set of vertices is the subgraph that results when only edges with both head and tail in the set are retained. In the first part of the function, marks are sent to each of the edges. An edge is marked to be saved if both its head and tail have marks. Function **induce-subgraph** uses a **pack** operation to sweep away edges that are not part of the induced subgraph.

The key step of the **MIS** algorithm is implemented by **select**. After computing the degree of each vertex, the function **coin** is used to probabilistically select vertices to be members of the independent set. Precisely, the function **coin** returns **true** if the degree of a vertex is $d = 0$, otherwise it returns **true** with probability $p = 1/(2d)$. The result of this step is a set of vertices, X, that have been selected to be in the independent set with a probability inversely proportional to their degree. (The implementation of **coin** has been omitted because it involves only element-wise arithmetic operations on a random value.)

Because this coin flipping procedure may select vertices that are not an independent set, some vertices must be removed from the selected set. Of vertices in the selected set, a vertex remains selected only if its degree is greater than the degree of any other adjacent selected vertex. Because each undirected edge of the original graph is stored twice in our bidirectional version, the function is written so that an edge de-selects its tail if the degree of the tail is less than (or equal to) the degree of the head. The field X represents the final selected set of vertices and is returned as the result of **select**.

The main procedure is called **MIS**. Fields I and Y are initialized so that no vertices are in the initial independent set but that all of them are in the currently active graph. In the main loop, the following actions occur. First, the typical mappings are created. Then the statement calling the coin flipping procedure, **select**, calculates a set of vertices from the active set, Y, that become the new candidates, II. This set is merged into I. The candidate vertices and all of their neighbors are then removed from the set of active vertices, Y. In the last step, edges that are not incident to active vertices are removed with **induce-subgraph**.

Since each of the ancillary functions only use mappings created in the main function so that their step complexities are all $S = O(1)$. Even **induce-subgraph** has an $S = O(1)$ step complexity. As explained earlier, because **pack** is implemented with a fixed-range **match**, its step complexity is $S = O(1)$.

For a graph with $n = |V|$ and $m = |E|$, Luby shows that with high probability this algorithm executes the main loop $O(\lg n)$ times. Each of the uses of **match** requires $O(\lg n)$ time so that the expected step complexity of the algorithm is $S = O(\lg^2 n)$. Our algorithm modifies Luby's by including the packing step in **induce-subgraph**. This adds nothing to the asymptotic step complexity of the algorithm but ensures that the problem size decreases as the algorithm proceeds. The upper bound on the work complexity of the algorithm is $W = O((m+n)\lg^2 n)$, but with the packing step the typical case is expected to be better in practice.

6.3 Minimum Spanning Tree

The minimum spanning tree (MST) problem is a classical network optimization problem. A network is a graph whose edges are each labeled with an associated real number called its weight. The minimum spanning tree problem asks for a subset of the edges that retains the connectivity of the original graph such that the sum of the edge labels is a minimum.

One of the earliest to study this problem and propose an algorithm to solve it was Boruvka, although Sollin rediscovered this algorithm later [Tar83]. Their algorithm is an iterative procedure that coalesces vertices into increasingly larger portions of the MST. Each portion is itself a small tree. At the completion of the algorithm, all vertices are part of the same MST if there is one.

At the beginning of the algorithm, each vertex is its own tree. The basis of their algorithm is the fact that for each tree, the minimum weight edge that joins it to another tree must be an edge in the MST of the entire graph. Thus, at each step, each tree simply finds the minimum weight edge adjacent to it. When all vertices are in the same tree, the algorithm is finished. It is interesting to notice that the algorithm is inherently parallel, as the minimum edge adjacent to each tree can be selected for all partial trees in parallel.

The same iterative procedure forms the basis the parallel minimum spanning tree algorithm designed by Savage and Ja'Ja' [SJ81], though their algorithm also makes use of techniques developed for the connected components algorithm of Hirschberg [HCS79]. Its clearest exposition, however, is given by Leighton [Lei92] whose formulation is presented here.

6.3.1 The MST Algorithm

Our algorithm represents the edges of the graph as a vector of tuples with fields <tail,head,W>, where W is the weight of the edge and the edge weights are assumed to be unique. Two main results are returned in fields of the vertex vector set. The first is the connected component that each vertex belongs to in field L. The second result is the set of edges of the MST in fields <tail,head>. Because there are only $(n-1)$ edges in the MST of an n vertex graph, these edges are placed in fields of the vertex vector. The nth edge of the MST is the root of the tree and is identified as the edge whose tail and head are the same vertex.

As Leighton pointed out, if it is not the case that the edge weights are unique, it is a simple matter to modify them so that they are. For a graph of n vertices, the weight of each edge,

(i,j), is modified so that $w' = (wn^2 + (ni) + j)$. Not only does this scheme ensure unique edge weights, it encodes the tail and head of each edge in its weight.

As the algorithm proceeds, it records information about the vertices in two fields called the leader, L, and parent, P, of each vertex. In this representation, the member vertices of a tree are identified by a distinguished vertex called its leader. All vertices belonging to the same tree have the same value in their L field. The leader of a tree is the vertex whose L value is the same as its index. If the graph is connected, then at the completion of the algorithm all vertices have the same leader and belong to the same tree. If the graph is not connected, then the leaders of each vertex identify the connected component to which each belongs.

The parent field, P, is used to link one tree onto another tree. By indicating how one leader links to another, these pointer values are used to implement the coalescing of trees. Parent pointers are discussed in detail later.

The entire MST algorithm is presented in Algorithm 6.6. The initialization phase, INIT, first sets each vertex to be its own leader and parent. Then, two mappings are created that are used to distribute values to the edges from their head and tail vertices. The rest of the algorithm consists of repeated application of two phases, the most important of which is the FINDMIN phase.

In the FINDMIN phase, each of the leaders selects the minimum weight edge that connects it to another tree. First, the current leaders are distributed to the head and tail of each edge using mappings Mtail and Mhead. These edge values i and j identify the trees bridged by each edge. If they are equal for some edge, then its endpoints are already in the same tree and it is not eligible to be added to the MST. Using this fact, a mapping is created from eligible edges to the leader of their tail. Ineligible edges use a **null** value in the **match** so that they are not considered. Note too, that vertices that are not leaders are not matched as a destination in the mapping either. Only eligible edges and current leaders participate in this step.

Using a **move-min**, the minimum weight edge connecting each leader to another tree is selected. This step makes use of a tuple to arrange the weight of an edge along with the leader of its head. Because the weight is in the most-significant place in the tuple, it will have precedence. This **move-min** step moves four values to each leader. They are

- The weight of the minimum weight edge connecting each leader to another tree.
- The leader of the other tree (it is moved into the parent (P) field).
- The head of the minimum edge.
- The tail of the minimum edge.

This single statement makes use of a powerful feature of tuples that allows multiple fields to be combined arithmetically. In fact, it implements the encoding of edge information in the weights of each edge in the same manner that Leighton suggests, but in a much clearer fashion.

```
mst(v.#, e.<tail,head>) {
  INIT:
    v.L = v.P = v.#;                                   //parent is leader is self
    Mtail = match(e.tail, v.#);
    Mhead = match(e.head, v.#);

    while(1) {
      FINDMIN:
        e.i = move-arb(Mtail, v.L);                    //label edges with current leaders
        e.j = move-arb(Mhead, v.L);
        if(and-reduce(e.i == e.j)) break;              //done if all in same component
        v.P = v.L;                                     //our current leader

        M = match(v.#, (e.i != e.j) ? e.i : NULL);     //match where not equal
        v.<W,P,head,tail> ?= move-min(M, e.<w,j,head,tail>);

      BREAKLOOP:
        v.PP = double(v.P);                            //our grandparent
        v.P = (v.PP == v.# & v.#<v.P) ? v.# : v.P;     //new LEADERs point at self

      CONTRACT:
        v.P = double*(v.P, v.#);                       //Contract trees

        v.L =move-arb(match(v.L, v.#), v.P);           //Update Leaders : L=P[L[i]]
    }

    where(v.L == v.#) {                                //roots terminate trees
        v.tail = v.head = v.#;
    }

    mst = v.<P,tail,head>;
}
```

Algorithm 6.6: Minimum Spanning Tree. Each pass through the main loop decreases the number of components by at least a factor of two. After $\lg n$ iterations all vertices are connected by the MST if it exists.

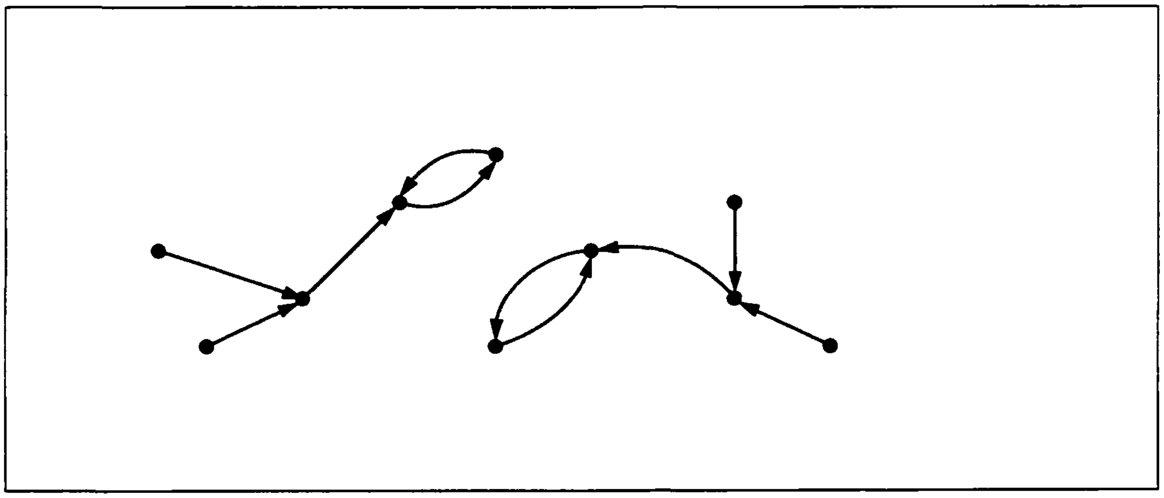


Figure 6.2: Tree Loops. Each vertex has an out-degree of one, and each component has a single cycle of length one. The root of each component is the vertex on the cycle with the lesser index.

When the BREAKLOOP phase is entered, the parent pointers of leader vertices describe a collection of tree loops as described in [HCS79]. (Because they were not matched, the parent pointers of non-leader vertices are not modified and have their previous values, which point to their leader.) A tree loop is a directed graph in which every vertex has out-degree one, and every component has a single cycle of length two. An example tree loop graph is shown in Figure 6.2. Using procedure **double**, each vertex checks to see if it is its own grandparent. If it is, then it is one of the two vertices on the loop and the lesser numbered vertex of the two is chosen as the root. The root vertex then changes its parent pointer to point to itself. The details of the correctness of this step are given in [Lei92]. After this step, the parent pointers describe simple rooted trees and they can be contracted using function **double*** as described in Section 4.5. Finally, in the last step, the leaders are updated to become the previous leaders of their new parents.

The algorithm terminates when the leaders of both ends of all edges are in the same tree. After exiting the main **while** loop, the tree edges connecting each vertex to its parent are returned in a tuple along with the leaders. If the graph is connected, then all leaders are the same. If they are not, then vertices with the same leaders are in the same connected component.

Storing the edges of the MST

The edges of the MST are left in the <head,tail> fields of the vertex vector. When a leader links onto another leader, the actual edge used is recorded in the <head,tail> of that leader. Notice that these fields are written to by the **move-min**. For each vertex, the last value written to these fields is the value written when a vertex changes its status from a leader vertex to a non-leader vertex — for as long as a vertex is a leader vertex it is eligible to be written to. Notice that the only vertices for which this does not happen on each step are those leaders that become the roots of a tree-loop, and hence, the leaders of the coalesced tree.

Thus, the values written in fields <head,tail> record for each parent link made the head and tail of the actual edge used to link each tree onto another. Upon termination of the algorithm, the only vertex that has not made the transition from leader to non-leader is the root vertex of the entire MST. This vertex is easily identified as the vertex that is its own leader. In the very last step of the algorithm, the <head,tail> fields of this vertex are set so that this vertex becomes the root of the MST.

Complexity

Each pass through the loop reduces the number of trees by at least half. In [SJ81], it is shown that this algorithm executes the main loop $O(\lg n)$ times for a graph of n vertices and m edges. The **match** performed in the FINDMIN phase requires $O(\lg n)$ steps and $O((n+m)\lg n)$ work, while the doubling step in CONTRACT performs $O(\lg^2 n)$ steps and only $O(n\lg^2 n)$ work. Thus, the step complexity of this algorithm is S=$O(\lg^3 n)$ and its work complexity is W=$O((n+m)\lg^3 n)$. The algorithm presented in [SJ81] required $t = O(\lg^3 n)$ time in the EREW PRAM model.

6.3.2 A better Minimum Spanning Tree Algorithm

Shiloach and Vishkin made an observation about the previous algorithm that allowed them to develop a better connected components algorithm [SV82]. In the basic algorithm, as leaders hook onto other leaders the parent pointers induce graphs on those vertices. Some of these graphs have larger diameters than others: in a sense, some are short and bushy, while others are long and stringy. The contraction of the parent-pointer graphs proceed quickly for those of small diameter while the large diameter graphs require more steps. While the larger diameter graphs are contracting, work is wasted on the small diameter graphs. The solution to this problem involves interleaving the FINDMIN stage and the iterations of the **double*** function of the CONTRACT stage.

In the new algorithm, shown in Algorithm 6.7, leaders are labeled as either *available* or *unavailable*. An unavailable leader is one whose tree is currently in the process of being attached to by another leader. The vertices of this other tree have yet to find their new leader. With a simple modification to the previous algorithm, Leighton showed how to use the observation of Shiloach and Vishkin to make a better MST algorithm [Lei92].

A field called avail is added for each vertex that is marked **true** if it is a currently available leader. Initially, all vertices are their own leader and are available. The FINDMIN phase is modified slightly so that only available leaders attempt to attach to another tree. This modification involves changing the source of the **match** step so that edges linked to unavailable leaders attempt

```
bettermst(v.#, e.<tail,head>) {
  INIT:
    v.L = v.P = v.#;                                  //parent is leader is self
    v.avail = true;                                   //all are initially available
    Mtail = match(e.tail, v.#);
    Mhead = match(e.head, v.#);

    while(1) {
      FINDMIN:
        e.i = move-arb(Mtail, v.L);                   //label edges with current leaders
        e.j = move-arb(Mhead, v.L);
        e.avail = move-arb(Mtail, v.avail);
        if(and-reduce(e.i == e.j)) break;             //done if all in same component
            //as before, but only include available leaders
        M = match(v.#, e.avail & (e.i != e.j) ? e.i : NULL);
            //Only leaders linking onto another tree are modified.
        v.<W,P,head,tail> ?= move-min(M, e.<w,j,head,tail>);

      BREAKLOOP:
        v.PP = double(v.P);                           //our grandparent
        v.P = (v.PP == v.# & v.#<v.P) ? v.# : v.P;    //new LEADERs point at self

      CONTRACT:
        v.P = double(v.P);                            //Contract trees only once
        v.L =move-arb(match(v.L, v.#), v.P);          //Update Leaders : L=P[L[i]]

      AVAIL: //Decide which superverts are now available
        v.PP = double (v.P);                          //find grandparent
        v.avail = (v.P == v.#) & move-and(match(v.#, v.PP), (v.PP==v.L));
    }

    mst = v.<P.T>;
}
```

Algorithm 6.7: A better Minimum Spanning Tree algorithm. This version interleaves the doubling of parent pointers with the linking of leaders onto other leaders. The result is an algorithm with a lower step complexity.

to match on a **null** key. Then, in the CONTRACT phase, only a single doubling step is performed (using **double** rather than **double***) when the leaders are updated.

Finally, each leader must decide if it is *available*. Once again, a doubling of the parent pointers brings each vertex its grandparent. A vertex j, is available if it is pointing to itself (it is a root) and if for all vertices whose grandparent is j, its leader is j too. Leighton writes this as:

$$P(j) = j \text{ and } P(P(i)) = j \Rightarrow L(i) = j \text{ for all } i.$$

This complex statement is translated into code using **match** and **move** in the AVAIL phase of the algorithm. It is instructive to study how **match** and **move** express this complicated mathematical formula succinctly and with ease.

Shiloach and Vishkin show that this approach also performs $O(\lg n)$ steps. Because the **match** and **double** operations require $O(\lg n)$ steps, the step complexity of this algorithm is only $S = O(\lg^2 n)$ in the vector model. The work complexity of this algorithm is $W = O((n+m)\lg^2 n)$. In the EREW model, the algorithm given by Shiloach and Vishkin had an $O(\lg^2 n)$ step complexity as well, so that our implementation using **match** and **move** is as efficient as theirs.

6.3.3 A better Connected Components Algorithm

The MST algorithm presented above may be used to find the connected components of a graph. Each edge (i, j) is assigned a weight $w(i, j) = i + nj$, where n is the total number of vertices. A minimum spanning tree is calculated along the way, but the storing of tree edges may be omitted. This algorithm has the same complexity measures as the previous MST algorithm: $S = O(\lg^2 n), W = O((n + m)\lg^2 n)$. Compared to the **simple-CC** algorithm of Section 6.1.5 with a work complexity of $W = O(n^3 \lg^2 n)$, this approach is much more work-efficient.

6.4 Series-Parallel Graphs

This section presents algorithms on a class of graph called Two-Terminal Series-Parallel (TTSP). TTSP graphs have two distinguished vertices called its "terminals", t1 and t2, and are described by the decomposition of their edges in terms of Series, S, and Parallel, P, reduction rules. The two rules are shown in Figure 6.3 along with an example TTSP graph. Each of the rules results in the reduction of the number of edges in the graph by one. By applying a sequence of these reduction rules, it is possible to reduce an TTSP graph to a single edge between two vertices. The following theorem defines the class of SP graphs.

A multigraph is TTSP if and only if it can be reduced to the trivial TTSP graph (two terminal vertices joined by a single edge) by a sequence of series and parallel reductions.

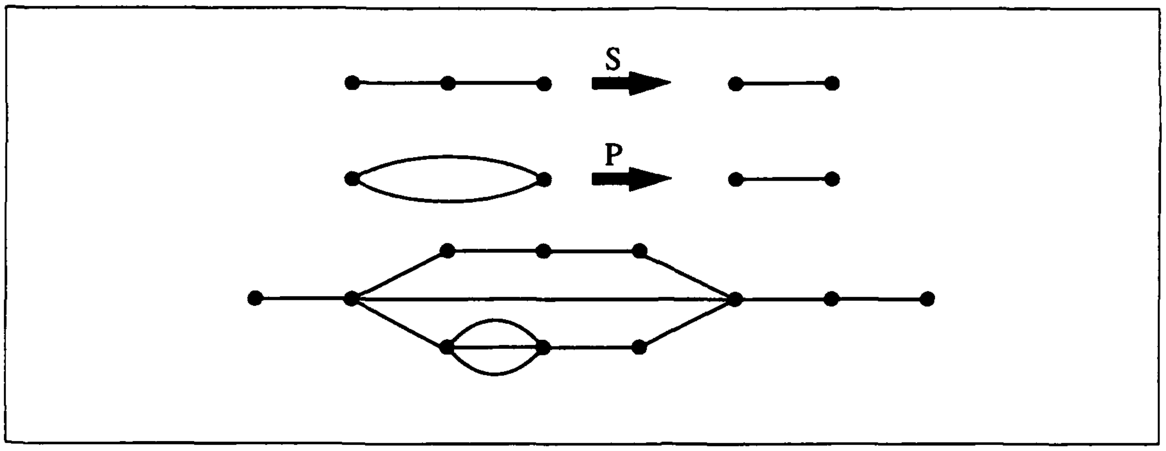


Figure 6.3: Series-Parallel Reduction Rules and an example Series-Parallel Graph. A graph is series-parallel if and only if it may be reduced to the trivial graph of two vertices and one edge by these rules.

This theorem is due to [VTL79] but is a trivial generalization of a corollary proposed by Duffin [Duf65]. Thus, the application of these reduction rules may be used to test if a graph is indeed TTSP.

There may be many different combinations of the rules that result in a single edge, all of which are valid reduction sequences for the graph. This particular set of reduction rules has been shown to possess the *Church-Rosser* property [Val78] which means that the order in which the rules are applied will not affect the possibility of reducing the graph to a single edge.

TTSP graphs are important because they occur naturally in the description of many different types of problems. Digital logic and switching circuits often exhibit TTSP structure as do many resistor networks. In these problems, edge labels on the original graph describe a quantity such the state of a switch or a resistance.

Through repeated application of the rules, an algorithm can be constructed that determines whether or not a graph is TTSP. In addition to recognizing TTSP graphs, the reduction rules may be used to implement algorithms on these graphs. For each rule, two edges are combined to produce one new edge. When series edges are merged, their labels are combined with some associative and commutative operator $\otimes$. Parallel edges are summed with $\oplus$, another associative and commutative operator. With suitable definitions of these operators, many different applications may be implemented. A table listing applications and the appropriate definitions of the operators is shown in Table 6.2.

6.4.1 The Parallel Reduction Rule

The edges in TTSP graphs are undirected. As before, an undirected graph explicitly stores both the forward and backward orientations of each edge in fields <tail,head,val>. This turns an undirected graph into a "bidirectional" graph. The label associated with each edge is stored in

Table 6.2: Problem Domains for Series-Parallel Graphs. The quantities that edge labels represent may be combined using different operators to solve different problems.

Problem Domain	Edge Quantity	$a \otimes b$	$a \oplus b$
Resistor Networks	resistance (ohms)	$a + b$	$1/(1/a + 1/b)$
Switching Networks	logical values	$a \wedge b$	$a \vee b$
Parsing TTSP	path expression	a CAT b	a UNION b

the val field. Each edge, (i, j), has a mate (j, i), and their labels must be the same. Thus, the minimal bidirectional TTSP graph has two vertices and one edge in each direction connecting them with the same label on each.

The recognition of parallel edges is trivial with **match**. Parallel edges are those with the same head and tail. Removal of redundant edges with **collapse** was shown earlier. The implementation of the parallel reduction step is shown in Algorithm 6.8. It is written with a generic **move-⊕** to indicate that edge values should be added together as appropriate for the problem being solved. One application of **parallel** creates a new graph with groups of parallel edges merged into one.

In this approach many parallel edges are reduced in one step. The arbitrary order in which the values are added by the **move-⊕** operator reflect the arbitrary order in which parallel reduction rules may be applied to a group of parallel edges. Because the $\oplus$ operator must be associative and commutative, this does not affect the end result computed. Notice too, that each new edge still has a mate with an equivalent label. Because all edges from i to j had a mate with the same label, the value computed for the new edge (i, j) is the same as the value for its mate, (j, i).

6.4.2 The Series Reduction Rule

Just as the parallel reduction rule could be extended to reduce more than two edges at a time, so can the series rule. A "chain" of series edges can be reduced in one step if all edges participating in a chain can be identified. By labeling the edges of each series chain the same, we can make use of **collapse** and **move-⊗** in a manner that is symmetric to their use in the parallel reduction rule.

While a series chain is easily recognized intuitively, a more formal definition is required. Vertices are characterized by their degree. Because the graph is bidirectional, the degree of each vertex in the undirected graph is the same as the in-degree of each vertex in the bidirectional graph. A vertex joining series edges is a non-terminal that has a degree of 2. Such a vertex is called a "chain" vertex.

```
parallel (e.<tail,head,val>) {
    M = collapse(e.<tail,head>);                        //recognition of parallel edges
    new.<tail,head> = move-arb(M, e.<tail,head>);
    new.val = move-⊕(M, e.val);
    parallel = new.<tail,head,val>;
}

series (e.<tail,head,val>, v.#, t1, t2){
    Min = match(v.#, e.head);
    v.d = degree(Min);

    e.td = move-arb(Mtail, v.d);                        //copy degrees to edges
    e.hd = move-arb(Mhead, v.d);
    //tail is a chain vertex iff degree=2 and not a terminal
    e.tchain? = (e.td == 2) & (e.# ! = t1) & (e.# ! = t2);
    e.hchain? = (e.hd == 2) & (e.# ! = t1) & (e.# ! = t2);

    Msucc = match(e.head, e.tail);                      //match successor
    e.<id1,other1> = move-max(Msucc, e.<#, head>);      //both successors of chain verts
    e.<id2,other2> = move-min(Msucc, e.<#, head>);
    e.next = (e.other1 == e.#) ? e.id2 : e.id1;         //select non-mate successor
    e.next = (e.hchain?) ? e.next : e.#;                //terminate chains
    e.key = double*(e.next, e.#);                       //label edges on a chain the same

    M = collapse(e.key);
    new.val = move-⊗(M ,e.val);
    new.head = move-max(M, (e.tchain?) ? 0 : e.tail);
    new.tail = move-max(M, (e.hchain?) ? 0 : e.head);
    series = new<tail,head,val>;
}

series-parallel(e.<tail,head,val>, v.#, t1, t2)
    while ( ! (done)) {
        size = length(e);
        e = parallel(e);
        e = series(e, v.#, t1, t2);
        done = (length(e) == 1) | (length(e) == size);
    }
    series-parallel = e.<tail,head,val>;
}
```

Algorithm 6.8: Series-Parallel graph reduction. The recognition of parallel edges is trivial. Series edges, however, represent a more difficult problem. The approach presented here recognizes series chains as linear linked lists and labels elements of each chain with the same key.

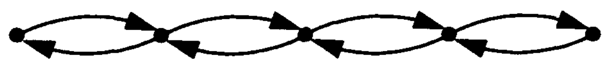


Figure 6.4: Series Chain in a Bidirectional Graph.

These two rules identify the edges of a chain in an undirected graph.

- An edge is at the end of a series chain if only one of its head or tail are a chain vertex.
- An edge is an internal chain edge if both its tail and head are chain vertices.

For each edge of a series chain in the undirected graph, there are two edges in the bidirectional graph. A fragment of a graph that contains a series chain is shown in Figure 6.4. These edges are linked into two directed chains that span the bidirectional edges of a chain by recognizing the following about bidirectional edges.

- An edge begins a directed series chain if its head is a chain vertex but its tail is not.
- An edge terminates a directed series chain if its tail is a chain vertex but its tail is not.
- An edge is an internal chain edge if both its tail and head are chain vertices.

Other edges do not participate in series chains at all.

Normally, when the words "successor" and "predecessor" are applied to directed graphs, they imply relations on the *vertices* of the graph. However, these relations also exist for edges. The set of "successors" of an edge (i, j) are those edges with tail j. This relation is easy to express by **matching** edges by tail to head. In a general bidirectional graph, note that one of the successors of each edge is its mate.

Edges in the vector representation are uniquely identified by their index. The strategy will be to recognize the successor of each edge that is the next edge in the directed series chain. A field called next will be used to store the index of the edge that follows in the chain, linking each edge to another. The next values will then describe simple rooted linked lists that can be contracted as shown in Section 4.5.1.

Directed series chains are now easily recognized by using the previous rules along with information about the successors of each edge. Because a beginning or internal chain edge has a head with degree=2, it has only two successors. The next edge in the series chain of such an edge is the successor that is *not* its mate. Other types of edges do not have a series chain successor, so their next field is directed to themselves. After these steps, the *next* field describes a rooted linked-list of edges that references each edge by its index in the edge vector. Because each list is uniquely identified by its root, the root of each list identifies a directed series chain in the graph. It is a simple matter to use of the **double*** function described earlier to find their root of each list so that all edges in a directed series chain are given the same key.

A diagram of the process is illustrated for a portion of a graph in Figure 6.5. First, the next relation is built linking each edge to the next in a chain, or to itself if it is not a beginning or internal edge. These decisions are made using the degree of the heads and tails of each directed edge. List-contraction then finds the root of each list.

This entire process is implemented in the code for the **series** reduction rule of Algorithm 6.8. In the first step, the degree of each vertex is computed. Then, using mappings Mtail and Mhead, degree information is moved to the heads and tails of each edge. In an element-wise fashion, each edge then calculates whether its head and tail are chain vertices and if they are not one of the two terminals, t1 or t2.

Next, a mapping is created that relates each edge to its predecessors. Using this mapping, two *distinct* successors are calculated for each edge. This step is done using the **move-min** and **move-max**. Because the index of each edge is unique, these two operations ensure that two different successors are returned for each edge. Elementwise, each edge can determine a successor edge that is *not* its mate. This step determines the correct next values for edges that are beginning or internal chain edges, but leaves other edges with incorrect values. These are corrected in the next step.

The next fields of edges that are not beginning or internal chain edges should simply point to them selves. In effect, all of these edges become roots of some linked list. For edges that terminate a series chain, they are a root of a multi-element list. All other non-series edges become the roots of lists of one element. This step is done in an elementwise fashion based on the degrees of the heads and tails of each edge. These values overwrite next values that were calculated for non-series edges in the previous step.

The next field describes a set of linked-lists that may be reduced to stars using **double***. By using the root of each list as a key for **collapse**, a mapping is created that moves all edges in a series chain to a common site. The labels of the new edges are then computed using **move-⊗** with this mapping. Because the graph is bidirectional, each undirected edge is represented twice, but the mates will have the same edge label.

There is one last matter of determining the tail and head vertices of the merged edges. Because series reduction removes the chain vertices, they must not become the heads or tails of the new edges. When moving the head and tail fields of each edge to the new sites, each vertex determines if it is a chain vertex. If it is, then the value 0 is sent. By using **move-max**, chain vertices are ignored and the head and tail of the new edge will be the head and tail of the chain.

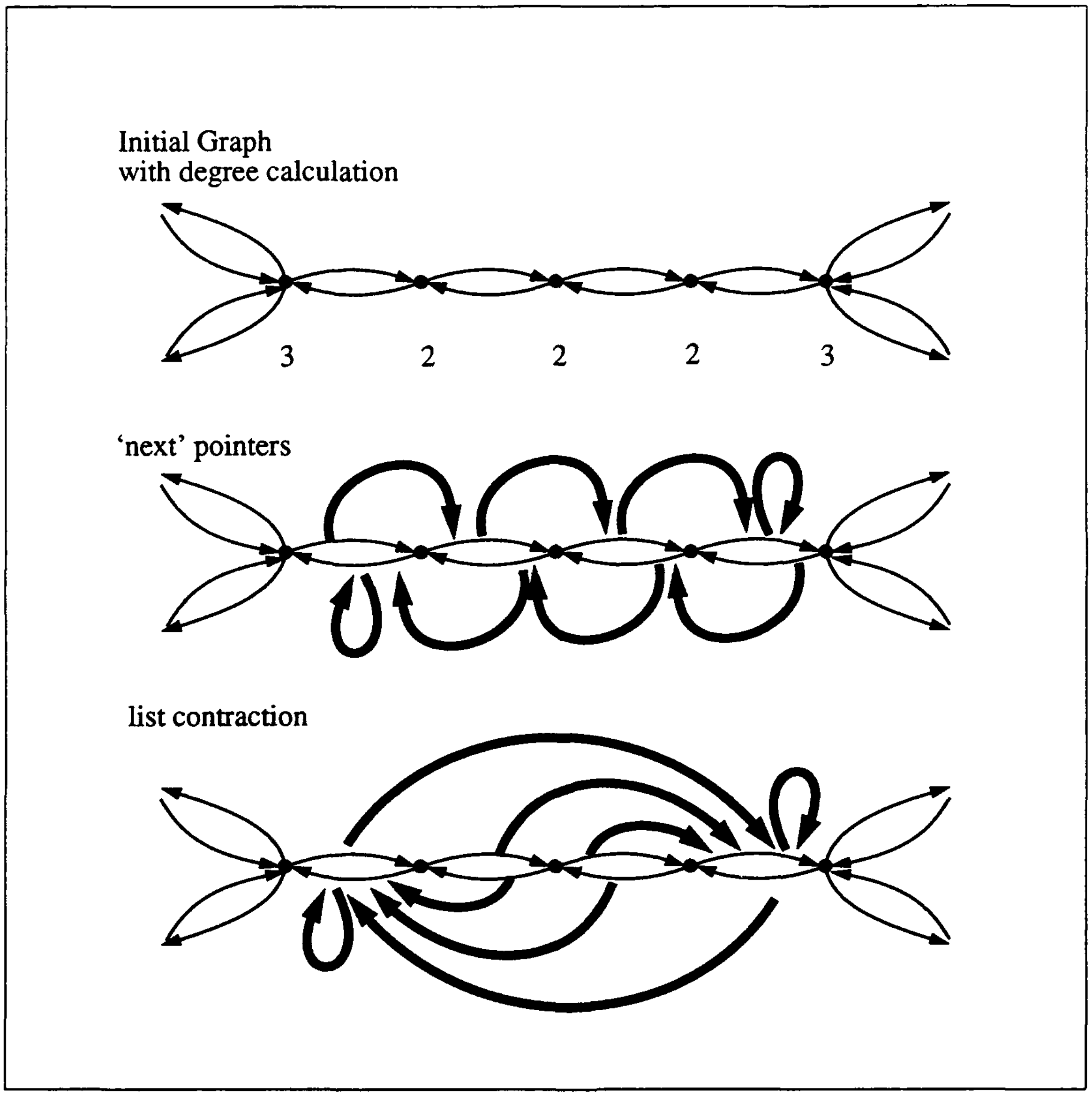


Figure 6.5: Recognition of edges in a chain and list contraction.

6.4.3 Series-Parallel Reduction

The reduction of an entire TTSP multi-graph is accomplished by repeated application of the two procedures, **parallel** and **series**. If the graph is reducible, then the process terminates when only one edge is left. If the graph is not a proper TTSP multi-graph, then at some point repeated application of the two procedures will not reduce the size of the edge set. This condition is easily detected at the end of the main loop.

For a graph of n vertices and m edges, the **parallel** reduction procedure has $S = O(\lg n)$ step complexity and $W = O((n+m)\lg n)$ work complexity. The step complexity is due to the use of **collapse**. The **series** reduction procedure creates four mappings with **match** and one with **collapse**. Each of these takes $O(\lg n)$ steps. The pointer doubling of the list-contract stage requires $O(\lg n)$ steps as well. Thus, the parallel reduction procedure has $S = O(\lg n)$ step complexity and $W = O((m+n)\lg n)$ work complexity.

Unfortunately, this parallel algorithm cannot guarantee that fewer than n applications of the reduction rules are required. In the worst case, a graph could require a strictly alternating sequence of the two, in which case $O(n)$ applications would be required. Thus, the step complexity of the entire TTSP recognition algorithm is $S = O(n \lg n)$. However, many interesting graphs do not exhibit this property. A better algorithm for the Series-Parallel recognition problem has been proposed by [HY87] that guarantees $O(\lg^2 n + \lg m)$ steps, but it is much more complicated than this one and only produces a decomposition tree of the graph.

6.5 Binary Tree Isomorphism

As described earlier, a forest of binary trees is represented as a vector of tuples. For each node, there is a tuple <id, left, right>. The id of each node is initially unique for each node. The fields left and right are references to the id fields of other nodes. Leaf nodes have the value **null** in their left and right fields. The binary tree isomorphism procedure computes new ID's for each node so that two nodes are given the same ID if and only if they are roots of isomorphic sub-trees.

This representation also supports Binary Directed Acyclic Graphs (DAG) which are similar to Binary Trees, except that there may be sharing of common sub-trees. The algorithm presented here determines isomorphic subtrees in Binary DAGs as well.

The algorithm shown in Algorithm 6.9 is a variant of the tree isomorphism algorithm of [AHU74]. A vector called t contains the forest represented through tuples. In the first steps, two mappings are created. The mapping Mleft relates left children to their parents, similarly, Mright is a mapping to the parents of right children. These two mappings freeze the structure of the forest as it is at the beginning of the procedure.

The procedure works iteratively. Initially, all nodes have different IDs. At each step, nodes

```
btree-iso(t.<id,left,right>) {
    Mleft = match(t.id, t.left);
    Mright = match(t.id, t.right);
    do {
        M = match(t.<left,right>, t.<left,right>);
        t.oldid = t.id;
        t.id = move-arb(m, t.id);
        t.left = move-arb(Mleft, t.id);          //Replace references of parents
        t.right = move-arb(Mright, t.id);
        t.done = (t.oldid == t.id);
    } until (and-reduce(t.done));
    btree-iso = t.<id,left,right>
}
```

Algorithm 6.9: Binary Tree Isomorphism. Label information propagates from the leaves to the roots. At the end of the algorithm, isomorphic sub-trees are guaranteed to be assigned the same label.

that are roots of isomorphic subtrees are identified by tuples with the same <left,right> values. Because all IDs are unique, in the first step, only leaf nodes whose <left,right> value are **<null,null>** will be given the same ID. Thus, after the first step, all subtrees whose height is 1 are labeled with the same ID if and only if they are isomorphic subtrees.

Now, proceeding inductively, at step number $h + 1$, all subtrees of height h have the same ID if and only if they are isomorphic subtrees. The root nodes of subtrees of height $h + 1$ are isomorphic if and only if their left and right children are isomorphic. The **match** operation on the <left, right> composite value of each node will create a mapping between the roots of all isomorphic subtrees, and the following **move-arb** will label all isomorphic subtrees of height $h+1$ with the same ID. For a forest of trees whose maximum height is H, the algorithm terminates after H steps, after information has been propagated up the trees from the leaves to the roots.

The complexity of each iteration is dominated by the **match** operation. Thus, for a tree of n nodes and height H, this procedure has step complexity $S = O(H \lg n)$ and work complexity $W = O(Hn \lg n)$. Because of these measures, this is not a work efficient algorithm, meaning that it performs more work than an equivalent serial counterpart.

6.6 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a data structure used for representing Boolean functions. While various forms of BDDs have been used for many years [Ake78], a recent refinement by Bryant has enhanced their appeal for many applications [Bry86]. Current research efforts have targeted BDDs for use in such areas as logic verification, test generation [Cho88], logic optimization and temporal verification of state machines [CMB90].

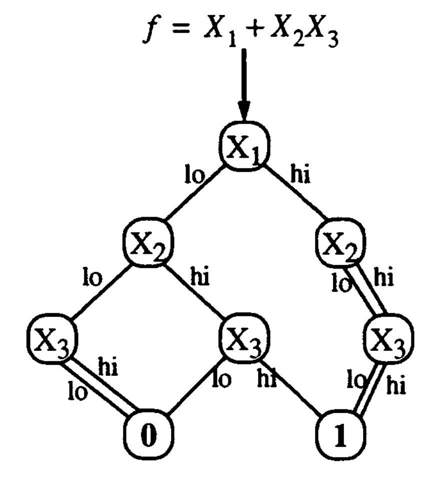


Figure 6.6: A sample BDD for the formula shown. The BDD represents a simple program for calculating the value of a Boolean formula based on the state of its variables.

A BDD represents a Boolean formula as a Directed Acyclic Graph (DAG) with two types of vertices: interior and leaf. A leaf vertex is labeled with one of the Boolean constants 0 or 1, and interior vertices are labeled with Boolean variables of the formula. Figure 6.6 shows an example of a BDD for the formula given.

A BDD may be interpreted as a simple program for computing the value of a Boolean function based on the values of its variables. Each interior vertex represents a decision to be made, choosing the LO child if the value of the variable is 0 or choosing the HI child for a value of 1. Tracing a path from the root, eventually a leaf vertex is reached and the value of the formula is computed as 1 or 0. Using the formulation of BDDs as state machines developed by Clarke [KC90], every path from a root to a leaf includes one node for each of the Boolean variables. When a total ordering is imposed on the variables of the formula so that in all paths traced to a leaf the ordering is not violated, the structure is called an Ordered Binary Decision Diagram (OBDD). This restriction leads to a canonical form for Boolean formulas represented as OBDDs.

The basic operation on OBDDs of this form is the function **apply**. It applies a Boolean operator such as AND or XOR to two formulas represented as OBDDs and computes the OBDD of the result. This operation is the basis for building OBDDs from simpler components. Before this function is introduced some mathematical background is required.

6.6.1 Shannon Expansion of Boolean Formulas and the apply Function

BDD construction follows directly from the Shannon expansion [Sha38] of Boolean functions [Bry86]. Briefly, the restriction of a Boolean function, f, when argument x_i is replaced by the constant b is denoted by $f|_{x_i=b}$. For a Boolean function f of arguments $(x_1, \cdots, x_m)$,

$$f|_{x_i=b} = f(x_1, \cdots, x_{i-1}, b, x_{i+1}, \cdots, x_m).$$

Using this notation, the Shannon expansion of a Boolean function about the variable x_i is given by

$$f = \overline{x_i} \cdot f|_{x_i=0} + x_i \cdot f|_{x_i=1}.$$

This equation is the basis for the OBDD representation. Each interior vertex in a OBDD represents a Boolean function f. Associated with each vertex is a variable x_i where the LO and HI children of the vertex are the restrictions of f, $f|_{x_i=0}$ and $f|_{x_i=1}$, respectively (see Figure 6.6).

Because the OBDD formulation is derived directly from the Shannon expansion of Boolean formulas, the definition of the **apply** function is closely related. The application of a Boolean operator $\langle op \rangle$ to formulas f_1 and f_2 of variables $x_1, \cdots, x_m$ is denoted by

$$\langle op \rangle [f_1, f_2](x_1, \cdots, x_m)$$

and is defined as

$$\langle op \rangle [f_1, f_2](x_1, \cdots, x_m) = \langle op \rangle [f_1(x_1, \cdots, x_m), f_2(x_1, \cdots, x_m)].$$

Applying the Shannon expansion yields

$$\begin{aligned} \langle op \rangle [f_1, f_2](x_1, \cdots, x_m) &= \overline{x_i} \cdot (\langle op \rangle [f_1|_{x_i=0}, f_2|_{x_i=0}]) + \\ &\quad x_i \cdot (\langle op \rangle [f_1|_{x_i=1}, f_2|_{x_i=1}]). \end{aligned}$$

This last form leads directly to a recursive formulation of a program for computing the OBDD for the application of an operator $\langle op \rangle$ to formulas f_1 and f_2 when f_1 and f_2 are represented by OBDDs. For each application of the rule, a new OBDD node is created for a variable x_i whose lo and hi children are the result of a recursive application of the rule. Notice that each subexpression on the right side of the equality is a recursive application of the rule involving the restriction of the original formula about a particular variable. If this variable is chosen correctly, then the restrictions are particularly easy to compute. Remember, that the functions f_1 and f_2 are represented as BDDs whose root nodes are labeled with variable x_1. By building the expansion about variable x_1 the necessary restrictions are simply the lo and hi children of the root nodes of the two formulas.

The recursive descent then proceeds with formulas involving only variables $(x_2, \cdots, x_m)$. As the recursion proceeds, a Depth First Search traversal of the two OBDDs is performed that creates a new node for pairs of nodes from the original formulas f_1 and f_2. The recursion ends when the Boolean functions of no variables (0 and 1) are reached at the terminal nodes, where $\langle op \rangle [f_1, f_2]$ may be computed directly.

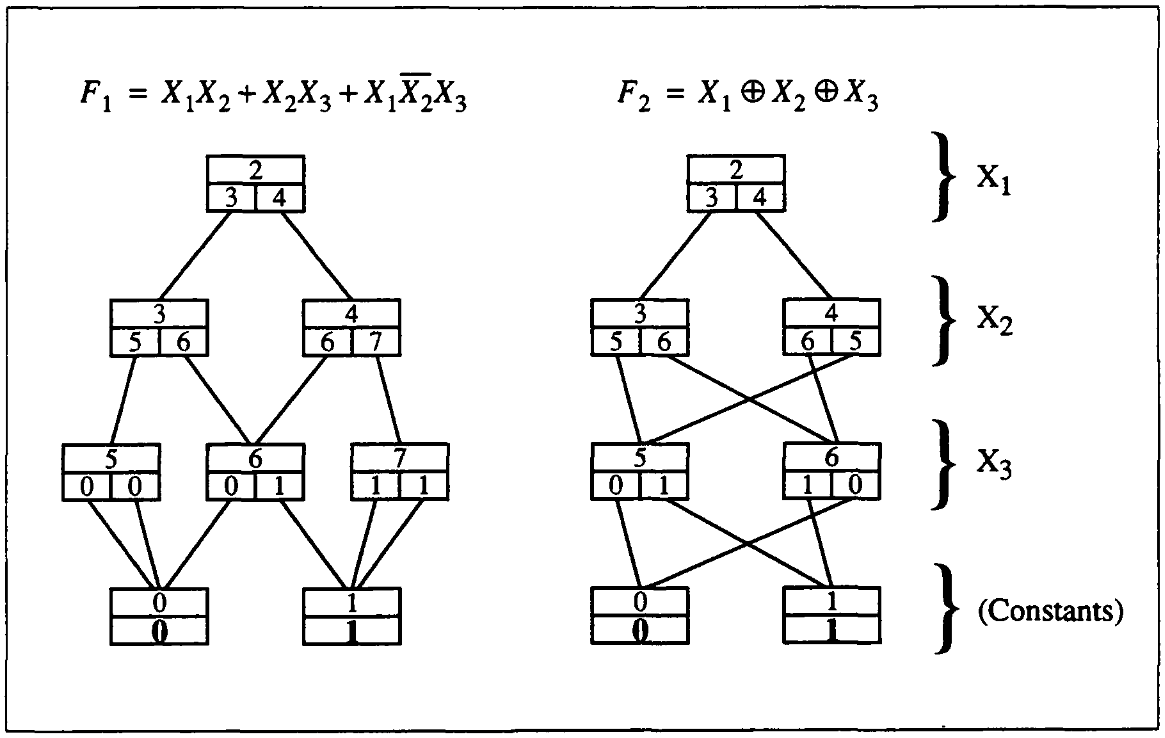

$F_1 = X_1X_2 + X_2X_3 + X_1\overline{X_2}X_3$
$F_2 = X_1 \oplus X_2 \oplus X_3$
X_1
X_2
X_3
(Constants)

Figure 6.7: Representing an OBDD in the vector model. The correspondence of a node id to a reference in the lo or hi field of another node indicates a parent/child relationship.

6.6.2 Parallel Binary Decision Diagrams

An OBDD is easily represented as a vector of tuples. Each node has five fields associated with it: <id,lo,hi,var,val>. The id, lo and hi fields serve the same purpose as the id, left and right fields of the binary tree introduced earlier. The var field labels each node with its variable and indicates its level in the OBDD. The val field is either 0 or 1 if the node is a leaf, or is an unknown value (X) for internal nodes.

An example of two formulas represented in this manner is shown in Figure 6.7. The var values are omitted from the nodes but are indicated by the level numbers of the variables. Internal nodes are shown with the values of their id, lo and hi fields, while leaf nodes are shown with their id and val fields.

In this example, the ids of the leaf nodes are 0 and 1. The *actual* ids of any of the nodes (including leaves) is irrelevant, but the leaves appear this way here to avoid confusion. Lines linking parents to children indicate a match of the id of a node to the lo or hi field of another node. It is this *matching* correspondence that determines the structure of the OBDD. The child fields of leaf nodes are null, and match no other ids.

The purpose of the val field is to allow the direct evaluation of a Boolean operator between two nodes of a OBDD. If both are leaf nodes, then the values involved are 0 and 1 and the result may be computed directly. Evaluation of the operator on an internal node with a value of X results in the unknown value. Such an unknown value indicates that the OBDD representation for the result formula is not a simple constant, **0** or **1**.

6.6.3 Parallel apply

In the recursive definition of **apply**, it is immediately apparent that every new node created is the product of a *pair* of nodes: one from the formula of f_1 and one from f_2. These nodes are uniquely identified with the tuple constructed from the ids of each of the constituent nodes: <id1, id2>.

The new lo and hi children of each node are similarly identified and each new parent node can create a reference to their new children by constructing the appropriate tuple pairs. This immediately suggests a parallel algorithm in which new nodes for all combinations of nodes from f_1 and f_2 are created, after which the references to the appropriate children are calculated. Unfortunately, this approach has the disadvantage that it will create a large number of new nodes that are not needed.

The key observation that avoids this pitfall (somewhat) is that new nodes are created only for pairs of vertices from the two formulas that have the *same* variable label. This implies a **match** between the nodes of the two formulas based on the var label. By creating only the pairs with the same variable label, the creation of many unnecessary nodes is avoided. This phase of the **bdd-apply** algorithm is implemented by function **bdd-flash** in Algorithm 6.10. The **match** and **move-join** create sites for all of the necessary pairs. Then, in an element-wise fashion, each of the new nodes calculates the tuple that becomes their new id, and the ids of the nodes that will become their left and right children. This process is called "Flash Expansion" and is illustrated in Figure 6.8 where all the necessary pairs have been created for the product of the formulas $(f_1 and f_2)$. Child pointers at this stage (lo and hi) are not absolute indices, but are simply references to the tuple id of another node. This process creates some unneeded nodes. For example, the nodes in the shaded region are unreachable from the root. They are dealt with in a later phase.

Of course, new nodes created this way have pairs of integers as their ids. One can easily imagine that repeated application of these types of nodes to **bdd-apply** would create increasingly larger ids. The function **bdd-normalize** is used to convert node ids back into simple integers. It first labels each node with its index in the new vector, and then uses the two parent-child mappings to update the child references with these new ids. This is called an OBDD in normalized form.

Even though the formulas combined using **bdd-apply** are in their canonical minimized form, it is not necessarily the case that the result computed by **bdd-flash** is too. First, there may be nodes that are not reachable from the root, and second, there may be isomorphic subgraphs in the structure. The first minimization is implemented by a top-down traversal of the BDD structure,

```
bdd-flash(OP, f1.<id,lo,hi,var,val>,f2.<id,lo,hi,var,val>){     //create new nodes in one step
    M = match(f1.var, f2.var);
    new.<i1,lo1,hi1,var1,val1,i2,lo2,hi2,var2,val2> =
            move-join(M, f1.<#,,lo,hi,var,val>, f2.<#,lo,hi,var,val>);
    elwise(new) {
        id = <i1,i2>;
        lo = <lo1,lo2>; hi = <hi1,hi2>;
        var = var1; val = val1 OP val2;
    }
    bdd-flash = new.<id,lo,hi,var,val>;
}

bdd-normalize(f.<id,lo,hi,var,val>) {                           //make each node's id its index
    Mlo = match(f.lo, f.id);
    Mhi = match(f.hi, f.id);
    f.id = f.#;
    f.lo = move-arb(Mlo, f.id);
    f.hi = move-arb(Mhi, f.id);
    bdd-normalize = f.<id,lo,hi,var,val>;
}

bdd-prune (f.<id,lo,hi,var,val>) {                              //remove unreachable nodes
    f.mark = (f.var == 1);                                      //initial label
    Mlo = match(f.id, f.lo);
    Mhi = match(f.id, f.hi);
    for (i = 1 to m) {
        f.mark |= move-or(Mlo, f.mark);
        f.mark |= move-or(Mhi, f.mark);
    }
    Mpack = pack(f.mark);
    bdd-prune = move-arb(Mpack, f.<id,lo,hi,var,val>);
}

bdd-reduce(f.<id,lo,hi,var,val>) {                              //merge isomorphic subtrees
    f.<id,lo,hi> = btree-iso(f.<id,lo,hi>);
    M = collapse(f.id);
    bdd-reduce = move-arb(M, f.<id,lo,hi,var,val>);
}
```

Algorithm 6.10: Helper functions for OBDD **bdd-apply**

```
bdd-apply(OP, f1, f2) { //apply a Boolean operator to two formulas
    f = bdd-flash(OP, f1,f2);
    f = bdd-normalize(f);
    f = bdd-prune(f);
    f = bdd-reduce(f);
    f = bdd-normalize(f);
    bdd-apply = f;
}
```

Algorithm 6.11: The OBDD **bdd-apply** function using the helper functions defined earlier

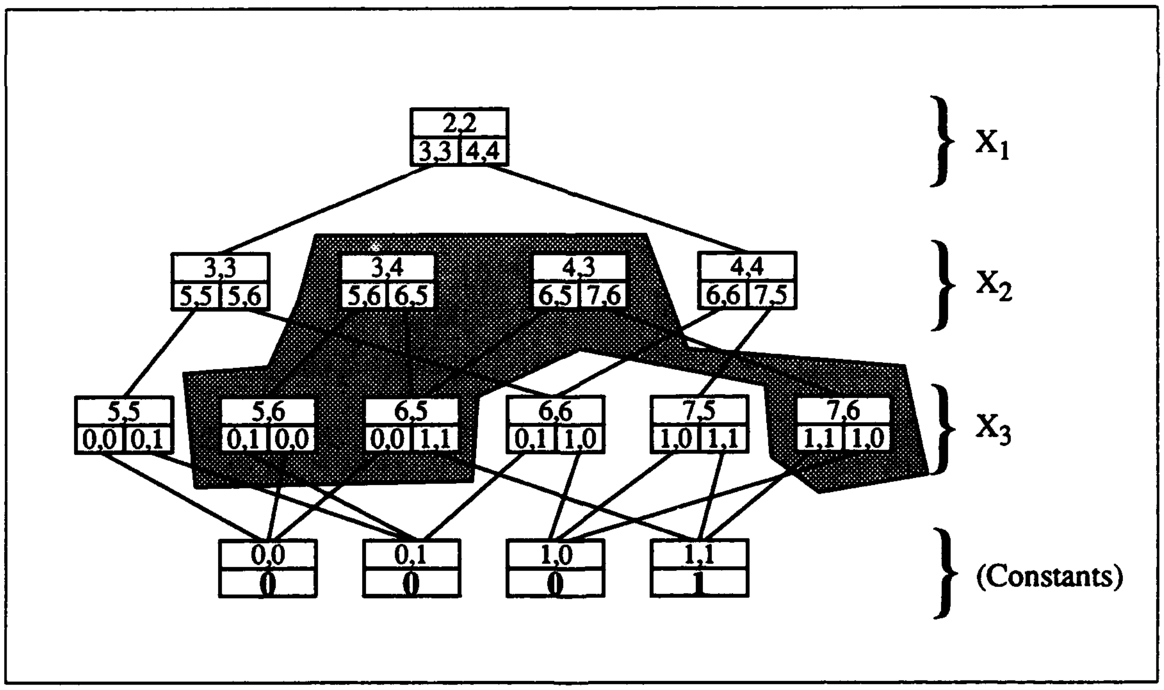


Figure 6.8: Flash Expansion of two OBDDs. In one step, all necessary pairs of nodes are created. However, extra garbage nodes are also created. The nodes in the shaded region are garbage because they are unreachable from the root.

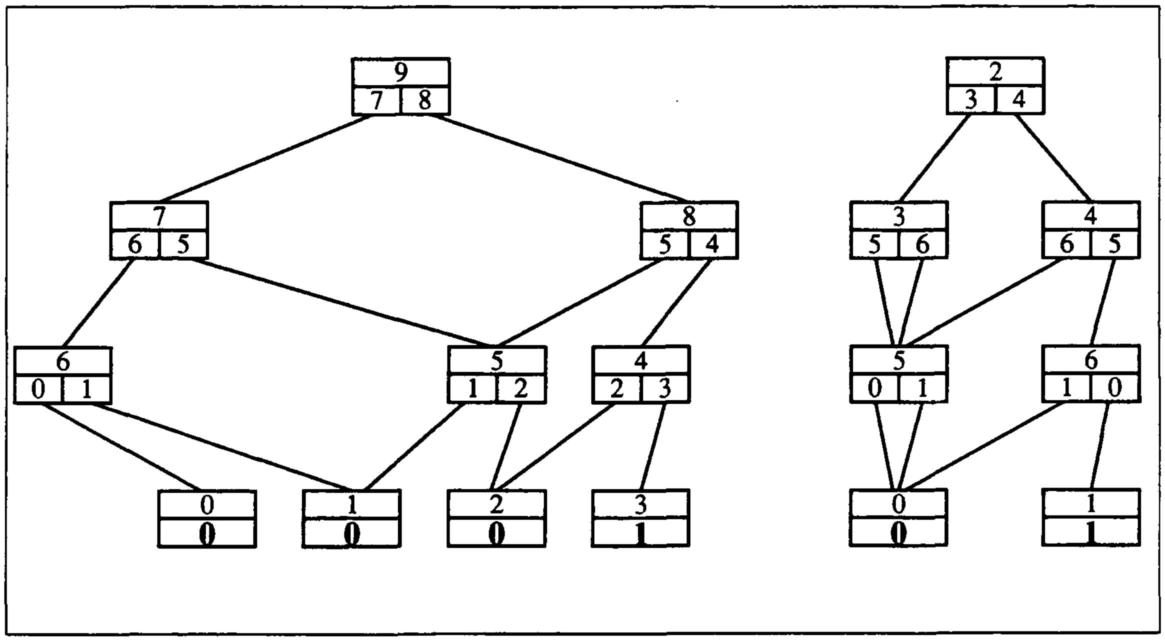


Figure 6.9: After **bdd-prune** and **bdd-reduce**. In the first OBDD, the unreachable nodes have been eliminated and the node ids have been normalized. The second OBDD shows the final structure after redundant nodes have been detected using the tree isomorphism algorithm.

marking only those nodes reached from the root. Those nodes that are not marked are removed by using **pack**. A function called **bdd-prune** implements this procedure. It initializes the mark field of each node to **true** only if it has the minimum valued variable number, in which case it is the root. Then, using the two mappings, marks are sent to the child of each root, with colliding marks combined using **move-or**. The process continues until all children have been marked. With an OBDD of m variables, this procedure will require m iterations.

There may still be isomorphic subtrees in the OBDD that represent the same Boolean formula. The binary tree isomorphism function of the previous section may be used to detect these and label nodes that are equivalent. Using **collapse**, duplicates are then swept away and another application of **bdd-normalize** puts the OBDD into normalized form. Snapshots of the OBDD after each of these two minimization steps are shown in Figure 6.9. After pruning unreachable nodes and normalizing their labels, the redundant nodes of the DAG are found by the tree-isomorphism algorithm. The end result is the OBDD that represents the product of the two original functions.

Using the component functions just described, the entire OBDD **apply** function is shown in Algorithm 6.11 using the component functions described. Table 6.3 lists the complexities of each of the functions in the vector model. The worst parts of this algorithm are the **bdd-prune** and **btree-iso** phases whose complexity is related to the depth of the tree. These work complexity measures could be improved by storing each level of the OBDD in a separate vector. This approach would also allow a level-by-level flash expansion that avoids creating unreachable

Table 6.3: Complexity of the BDD **bdd-apply** function in the vector model. The number of nodes in each of the two arguments to **bdd-flash** are given by f_1 and f_2. The other functions take only a single argument with a node count of f. Parameter m is the total number of variables or levels.

Function	Step	Work
flash	$O(\lg(f_1 + f_2))$	$O(((f_1 + f_2)\lg(f_1 + f_2) + (f_1 f_2))$
normalize	$O(\lg f)$	$O(f \lg f)$
prune	$O(\lg f + m)$	$O((\lg f + m)f)$
reduce	$O(m \lg f)$	$O(mf \lg f)$

nodes. However, these modifications would have obscured the simplicity intended for this tutorial example.

Chapter 7

Sparse Matrix Algorithms

Sparse matrix applications are generally considered ill-suited for parallel machines. Because sparse matrices present an irregular data structure with no direct mapping onto parallel hardware, they represent a difficult algorithm design problem. Sparse matrix algorithm construction using **match** is simple and elegant.

The representation of a sparse matrix is as a vector of <col,row,val> tuples, which is very similar to the storage of edges in a graph as <tail,head,val> tuples. The relationship between graphs and sparse matrices is a familiar one. For a matrix M, define $G(M)$ to be the graph $G = \langle V, E\rangle$ associated with M such that if M is of order n, there are vertices v_i, $0 \leq i < n$, and that for each non-zero entry $M_{j,i}$ (row j and column i) there is an edge (i, j) with tail v_i and head v_j. The value of each edge, $e(i, j)$, is the value of the matrix entry $M_{j,i}$. This definition is not appropriate for non-square matrices, but by viewing sparse matrix operations as graph algorithms anyway, it is easy to write procedures for them using **match** and **move**.

7.0.4 Sparse Matrix-Vector Multiplication

The multiplication of a dense vector by a sparse matrix, $x = Ab$, has an appealing graphical representation. The process is illustrated in Figure 7.1. Each vertex, v_i, is given an associated value $b(i)$. The matrix-vector product is computed in two phases. First, the value of each vertex is sent to each of its outgoing edges, where it is multiplied by the value of the edge. The result vector, x, is computed by summing the products on the incoming edges of each vertex, The code for function **mvmult** is given in Algorithm 7.1.

The more traditional view of matrix-vector multiplication distributes the elements of the vector along the columns of the matrix where the products are computed. Then, the non-zero values along the rows are summed. The two phases of this algorithm correspond directly to this process. Because columns in the matrix correspond to tail vertices in the graph, the first mapping distributes the vector elements over the columns. Similarly, the second mapping gathers elements

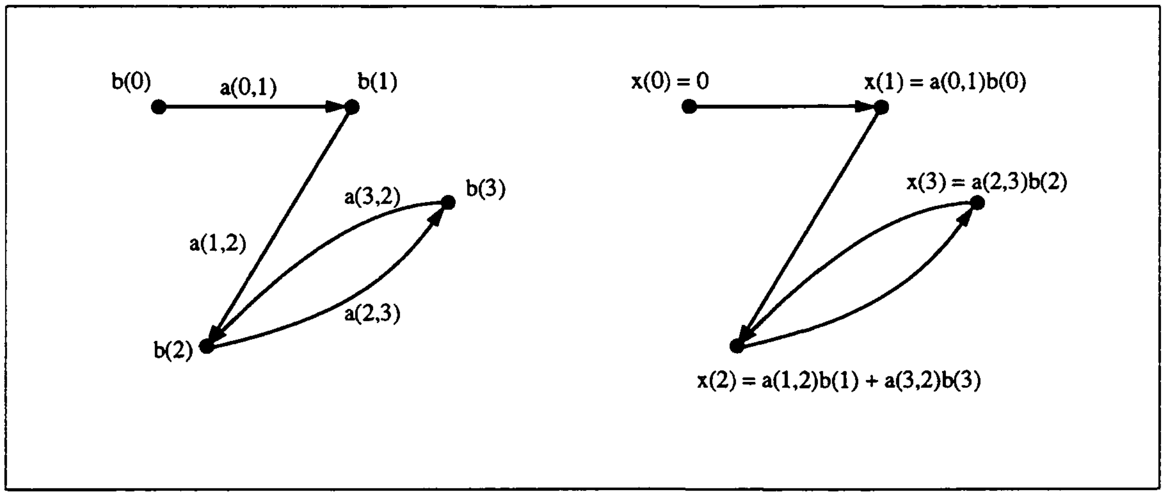


Figure 7.1: The graphical interpretation of $x = Ab$. The value computed for each vertex is the sum of each of its neighbors being scaled by the edge that connects them.

```
mvmult(m.<col,row,val>, v.val) {                    //matrix-vector multiplication
    Mtail = match(m.col, v.#);
    Min = match(v.#, m.row);
    m.copy = move-arb(Mtail, v.val);                //distribute values across out-edges
    m.prod = m.copy × m.val;                        //calculate all products
    mvmult = move-add(Min, m.prod);                 //sum along in-edges (rows)
}

mmadd(A.<col,row,val>, B.<col,row,val>)             //matrix-matrix addition
    t = append(A, B);
    M = collapse(t.<col,row>);                      //merge parallel edges
    new.<col,row> = move-arb(M, t.<col,row>);
    new.val = move-add(M, t.val);
    mmadd = new.<col,row,val>;
}

mmmult(A.<col,row,val>, B.<col,row,val>) {          //matrix-matrix multiplication
  PHASE1:
    M1 = match(B.row, A.col);
    pp.<col, Bval row, Aval> = move-join(M1, B.<col,val>, A.<row,val>);
    pp.val = pp.Bval × pp.Aval;
  PHASE2:
    M2 = collapse(pp.<col,row>);
    new.<col,row> = move-arb(M2, pp.<col,row>);
    new.val = move-add(M2, pp.val);
    mmmult = new.<col,row,val>;
}
```

Algorithm 7.1: Basic sparse matrix algorithms.

on the same row so that they may be summed using **move-add**.

The functions **degree** and **neighbor-sum** described earlier are more specific versions of this general procedure. When calculating the degree of each vertex, the b values are all 1, as are the edge values. The neighbor-summing algorithm simply assumes that all of the edge values are 1. This basic operation appears in other forms in other algorithms as well. For example, in maximal flow network algorithms, **max** is substituted for the multiplication operator. Later, a number of different operations over different semirings are unified in the discussion of Sparse Gaussian Elimination.

For a graph with n vertices and m edges, the complexity measures associated with this function are $S = O(\lg n)$ and $W = O((m+n)\lg n)$.

7.0.5 Sparse Matrix-Matrix Addition

Sparse matrix addition is especially straightforward using this representation. Given sparse matrices A and B, the matrix representing their sum has as non-zero elements the union of the elements of A and B. Where a non-zero element from A matches an element from B, they are added together.

The matrix addition procedure proceeds much as set **union** did earlier. The function **mmadd** is shown in Algorithm 7.1. After merging the two sets of elements into one vector, matching values are identified as those with the same composite key <col,row>. Matching elements are then added together by using **collapse** and **move-add**.

This function has the same complexity measures as **union**. Because a **match** on the edges of any graph of n vertices requires $O(\lg n)$ steps, even when two matrices are involved, the step complexity of the algorithm is $S = O(\lg n)$. The work complexity is related to the sum of the lengths of the two edge vectors: $W = ((m_1 + m_2)\lg n)$.

7.0.6 Sparse Matrix-Matrix Multiplication

Given two matrices A and B, the matrix product, $C = A \times B$, is defined as follows. An element of matrix A in row i and column j is denoted $A_{i,j}$. Each element of the matrix product C is computed by the summation

$$C_{i,j} = \sum_{1 \le k \le N} A_{i,k} B_{k,j}.$$

When A and B are sparse, it is advantageous to avoid storing elements that are zero and computing products where one of the operands is zero. Using **match**, the algorithm consists of two parts. The first phase computes all of the partial products that contribute to the sums, along with their indices in the result matrix, C. The second phase gathers partial products with the same indices in the result matrix and adds them together.

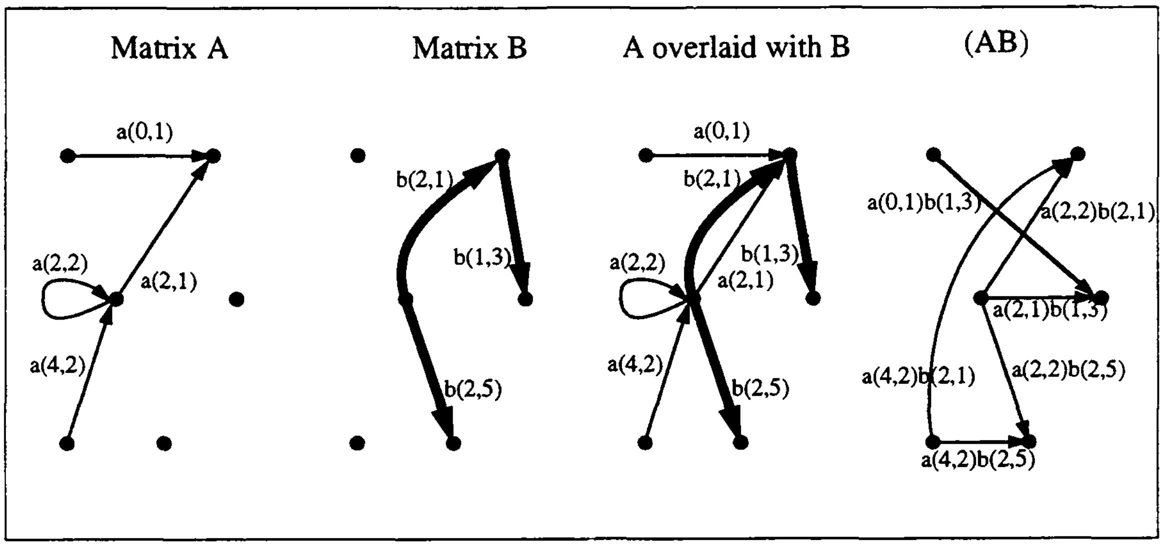


Figure 7.2: The graphical interpretation of Sparse Matrix-Matrix Multiplication. A new edge is created where the head of an edge from A meets the tail of an edge from B. This procedure is nearly identical to graph squaring, except that the edges are selected from two distinct classes.

In examining the formula above, each product is of the form $A_{i,k}B_{k,j}$, indicating a "match" of the columns of matrix A with the rows of B. The partial products created between a column of A that matches a row of B are all of those that result when the elements from a column k of A are distributed over the elements of row k of B. A simple application of **move-join** to the mapping creates sites for all of these partial products. The implementation of **mmmult** is shown in Algorithm 7.1.

The graphical representation of this operation is similar to the graph squaring operation shown earlier. For two matrices A and B, the elements of the two matrices are superimposed over one another as two different types of edges in one graph. In the example of Figure 7.2 edges from the B matrix are shown with bolder lines than those of the A matrix. Product edges are created wherever edges of the two types meet such that a head of and edge from A meets the tail of an edge from B. Once again, this process typically creates multiple edges between pairs of vertices that must be added together using **collapse** and **move-add**.

When multiplying two matrices, the maximum number of products that could be computed is n^3. Thus, the step complexity of the algorithm is determined by performing a **match** on that many edges, so $S = O(\lg n)$. In the very worst case, the work complexity is $W = O(n^3 \lg n)$, however because the matrices are sparse it should be much less in practice.

7.1 Sparse Gaussian Elimination

Gaussian elimination is a well-known algorithm most often used to solve systems of linear equations. However, it can also be used to solve a more general set of path problems on directed graphs [Tar81]. These include finding shortest paths, determining reachability, and calculating largest-capacity and most reliable paths. Each of these problems is described as a system of equations over a suitable semiring, and the Gaussian elimination procedure is applied with definitions of addition and multiplication appropriate for the problem.

Following the terminology of [AS85], a semiring is a system $(S, +, \times, ^*, \mathbf{0}, \mathbf{1})$ in which S is a set closed with respect to the binary relations + (addition) and $\times$ (multiplication) and the unary operation *. The symbols $\mathbf{0}$ and $\mathbf{1}$ are elements of S and the usual laws of associativity, additive commutativity, distribution and additive and multiplicative identity are satisfied. In this formulation, the * operation satisfies

$$a^* = aa^* + 1 = a^*a + 1.$$

In the literature, this operation is called either "closure" or "asteration." Notice that the operation is defined implicitly, as the solution of an equation. However, for the most interesting domains, there are closed form expressions for its evaluation.

The formulation of each of these path problems on graphs is as a system of equations in the form

$$Ax + b = x.$$

The matrix A describes the connectivity of the graph, b is a vector of constants that initially label the vertices, and x is a vector of unknowns that represents the solution of the system. The same Gaussian elimination algorithm may be used to solve any of the path problems on graphs by interpreting the operations over a suitable semiring. From [AS85], a list of some types of semirings and the problems they solve are shown in Table 7.1.

Gaussian elimination is presented here as a graph algorithm. Although there is a wealth of literature about the relationship between Gaussian elimination and graph operations, it is most often viewed as a matrix algorithm. A graph representation is convenient for representing the non-zero structure of A when A is sparse. While such graphs have been used to assist in the selection of an elimination ordering for performing Gaussian elimination of systems of linear equations [Gil80, Ros72, RT78], with an appropriate labeling, the graph itself may be used to directly solve the system of equations.

Each vertex in the graph has two associated labels: $b(i)$ and $x(i)$. The labeling $b(i)$ corresponds to the vector of constants b in the matrix formulation of the problem, and the labeling $x(i)$ to the solution of the system. The basic serial Gaussian elimination algorithm is presented in a graph theoretic framework in Algorithm 7.2. Given an initial labeling of the vertices b this algorithm calculates the solution of the system in the labels x.

The algorithm may be understood as the propagation of information through the graph. At each step, a vertex is selected for elimination from the graph. Before doing so, its label is propagated to each of its neighbors after scaling by the appropriate edge values. This process is

Table 7.1: Some examples of semirings for path problems on graphs.

S	$a+b$	$a \cdot b$	a^*	0	1	Description	Application
$\{0,1\}$	$a \vee b$	$a \wedge b$	1	0	1	Boolean Values	Reachability in graphs; reflexive and transitive closure of binary relations
$R \cup \{\infty\}$	$\min(a,b)$	$a+b$	0	∞	0	Real numbers augmented with the element ∞	Shortest paths
$R^+ \cup \{\infty\}$	$\max(a,b)$	$\min(a,b)$	∞	0	∞	Nonnegative real numbers augmented with the element ∞	Largest-capacity paths
$[0,1]$	$\max(a,b)$	ab	1	0	1	Real numbers between 0 and 1 inclusive	Most reliable paths
$R \cup \{\infty\}$	$a+b$	ab	$\frac{1}{1-a}$ $a \neq 1$, $1^* = \infty^* = \infty$	0	1	Real numbers augmented with the element ∞	Solution of linear equations

```
FORWARD-ELIMINATION:
for i = 1 to $n$ do
    for j = $i$+1 to $n$ do
        $b(j)=b(j)+b(i)a(i,i)^*a(i,j)$                //propagate labels to neighbors
        for k = $i$+1 to $n$ such that $(k,i) \in E$ do
            if $(k,j) \in E$ then                     //is this a fill-in edge?
                $a(k,j)=a(k,j)+a(k,i)a(i,i)^*a(i,j)$
            else
                $E=E \cup \{(k,j)\}$                  //add edge structure
                $a(k,j)=a(k,i)a(i,i)^*a(i,j)$
            fi
        od
    od
od

BACK-SUBSTITUTION:
for i = $n$ downto 1 do
    for j = $n$ downto $i$+1 such that $(j,i) \in E$ do
        $b(i)=b(i)+x(j)a(j,i)$                        //backwards propagation of labels
    od
    $x(i)=b(i)a(i,i)^*$
od
```

Algorithm 7.2: The basic serial Gaussian elimination algorithm. This graph-oriented presentation of a familiar algorithm exploits sparsity explicitly.

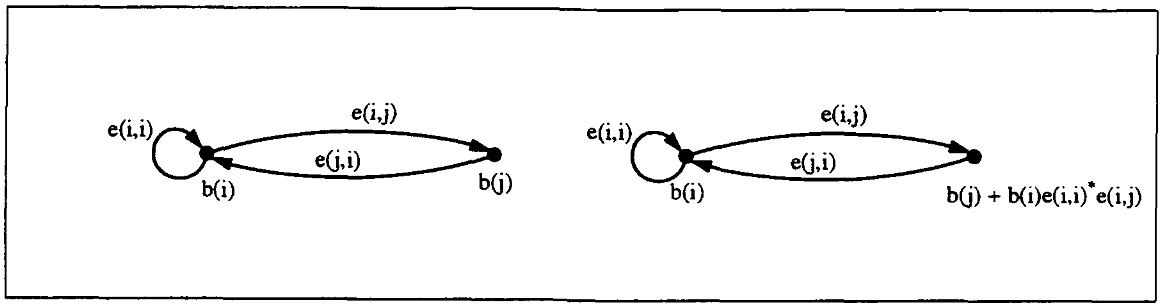


Figure 7.3: Forward Propagation in Gaussian Elimination for a vertex i. Values are propagated only over those edges emanating from a selected vertex.

illustrated in Figure 7.3. Notice that the value of the self-directed edge on the eliminated vertex plays an important role.

Then, new edges are computed between each pair of neighbors of the eliminated vertex. These are called the "fill-in" edges, as they possibly add new edges to the graph. This process is illustrated in Figure 7.4. A new edge value is produced for each pair of incoming and outgoing edges of the eliminated vertex. Where fill-in edges correspond to existing edges, their values are added to the existing edge. For pairs of edges to and from the same neighbor, a new self-directed edge value is added to the neighbor.

To continue the metaphor of information flowing through the graph, both the neighbor-update and fill-in steps propagate information about the eliminated vertex to its neighbors. The neighbor-update step sends its value, while the fill-in step maintains connectivity information that the vertex provided between its neighbors. This is the intuition behind how transitivity information is propagated through the graph.

The basic update operation of the backsubstitution process is similar to that of the forward propagation step. During backsubstitution, vertices are successively added back into the graph along with their incident edges in the reverse order of their elimination. As each is added, its label is updated from the values of all of its neighbor vertices. These are the same neighbors that it had at the time it was eliminated. Continuing the information flow metaphor, this step provides the propagation of information back to the early vertices about vertices and edges eliminated after them. The final solution of the system is calculated after all vertices have been added back to the graph.

The creation of fill-in edges is the factor that determines the efficiency of the procedure for sparse graphs. For certain classes of graphs, there are good vertex elimination orderings that maintain the sparsity of the graph [Geo73]. Sometimes, these can guarantee linear complexity of the procedure with respect to the number of vertices. In contrast, a bad elimination ordering can introduce edges between almost every pair of vertices, resulting in an $O(n^3)$ algorithm. In general, the selection of an optimal elimination ordering is NP-complete [Yan81, GJ79].

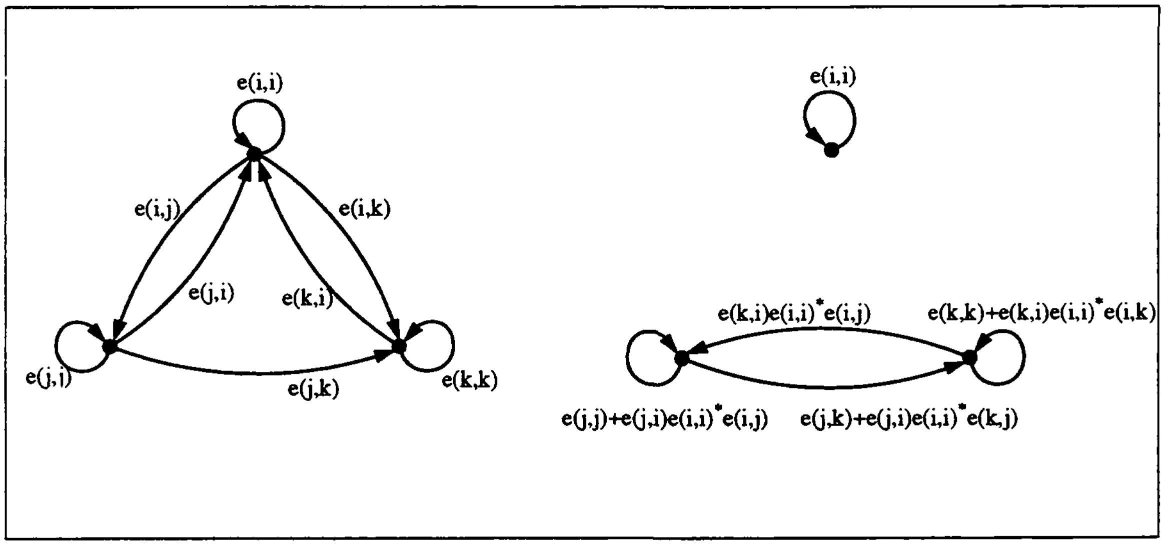

e(i,i)
e(i,j)
e(i,k)
e(j,i)
e(k,i)
e(j,j)
e(j,k)
e(k,k)
e(i,i)
e(k,i)e(i,i)*e(i,j)
e(k,k)+e(k,i)e(i,i)*e(i,k)
e(j,j)+e(j,i)e(i,i)*e(i,j)
e(j,k)+e(j,i)e(i,i)*e(k,j)

Figure 7.4: Fill-In Edge Creation in Gaussian Elimination. A new edge is created for every in-going/out-going edge adjacent to a vertex selected for elimination. The values of the fill-in edges are computed from the two edges, as well as the self-directed edge of the eliminated vertex.

7.1.1 Parallel Gaussian Elimination

The basic Gaussian elimination algorithm may be parallelized by taking advantage of opportunities to eliminate multiple vertices simultaneously. At each step, a number of vertices may be eliminated in parallel if they are an independent set. This observation forms the basis of the parallel elimination scheme. This differs significantly from such methods as Parallel Generalized Nested Dissection [PR85] where a small set of vertices is eliminated in parallel even if they are not an independent set.

Parallel elimination complicates the implementation of the algorithm. First, in the neighbor update phase, if two or more vertices selected for elimination have a neighbor in common, then the neighbor will receive update values from all of the eliminated vertices. Secondly, when creating fill-in edges, multiple fill-in values may be created for different pairs of vertices. Figure 7.5 shows both of these situations for three vertices selected for parallel elimination. The vertices selected are shown circled. Elimination of the vertices in parallel will require multiple values to be added into the other two vertices. Furthermore, the fill-in values that result will add multiple edges between these same two vertices. What these two operations have in common is that the values computed for the same vertex or edge must be merged using addition. Using **match**, corresponding values are gathered together and are then combined using **move-add**.

7.1.2 Factor and Solve

The original Gaussian elimination algorithm was formulated as a single procedure. The parallel Gaussian elimination algorithm, broken into procedures **factor** and **solve**, appears in Algo-

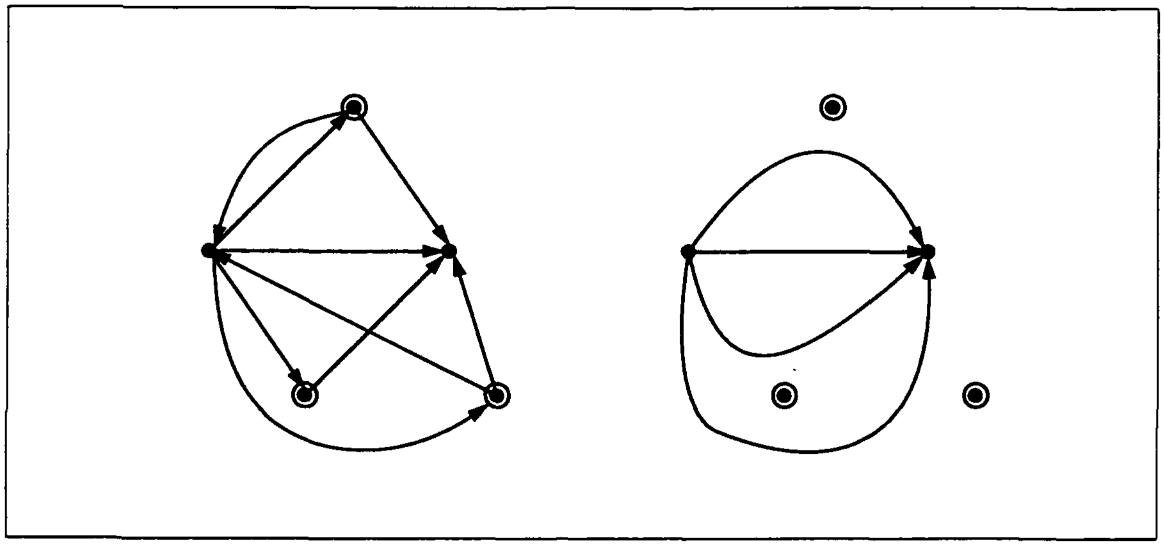


Figure 7.5: Parallel Gaussian Elimination. The circled vertices may be eliminated in parallel because they form an independent set. Their elimination modified the values of the other two vertices, and results in the fill-in edges shown on the right.

rithm 7.3. Procedure **factor** performs the operations on the edges, while **solve** finds a solution for a particular b vector. This two-part decomposition is useful when the same system of equations (matrix A) must be solved for multiple right-hand sides (vector b).

Procedure **factor** loops until all vertices have been eliminated. A counter called elimnumber keeps track of the total number of parallel elimination steps. At each step, the function **mark** selects a suitable set of vertices that is an independent set. (The discussion of how this is done is deferred for now.) Each of the vertices selected is marked **true** in their sel (selected) field. The next steps categorize each of the edges according to the way their endpoints are marked. If only the tail is marked for elimination, then the edge is a *forward* edge. A *backward* edge has only its head marked. An edge with both its head and tail marked must be a self-directed edge because the selected vertices form an independent set. Edges with no marks are not involved in the current elimination step.

These four types of edges are moved to four vectors by using the **pack** function. This step represents a partitioning of the current matrix into four separate parts whose union is the original matrix. Note that the vector approach to this operation using **pack** simplifies a partitioning operation that would be difficult to express in conventional matrix notation.

As shown in Figures 7.3 and 7.4, the asterates of the self-edges of the eliminated vertices have a special function in both the forward propagation step and the fill-in step. These values are first moved to the ast field of each selected vertex where they are then used to scale their forward edges. Because the asterate values multiply the forward edges in both the forward propagation step and fill-in edge calculation, it is easier to simply scale the forward edges once

```
factor(e.<tail,head,val>, v) {
    elimnumber = 0;
    while(length(e) ! = 0) {
        v.sel = mark(e.<tail,head,val>, v.#);                    // selected for elimination
        e.telim = move-arb(Mtail, v.mark);
        e.helim = move-arb(Mhead, v.mark);

        //Break edges into categories
        fwd.<tail,head,val> = move-arb(pack(e.<telim,helim>==<1,0>), e.<tail,head,val>);
        bak.<tail,head,val> = move-arb(pack(e.<telim,helim>==<0,1>), e.<tail,head,val>);
        self.<tail,head,val> = move-arb(pack(e.<telim,helim>==<1,1>), e.<tail,head,val>);
        other.<tail,head,val> = move-arb(pack(e.<telim,helim>==<0,0>), e.<tail,head,val>);

        //save asterations of self-edges of selected verts
        v.ast ?= asterate(move-arb(match(v.#, self.head), self.val));

        //Scale FWD edges by AST of tail self-loop
        fwd.val ×= move-arb(match(fwd.tail, v.#), v.ast);

        //save fwd and bak edges for solve
        FWD[elimnumber] = fwd.<tail,head,val>;
        BAK[elimnumber] = bak.<tail,head,val>;

        fill = mmmult(bak, fwd);                                 //create new edges
        e = mmadd(other, fill);                                  //merge into existing edges
        elimnumber++;
    }
}

solve(v.<b,x>) {
  FORWARD-PROPAGATION:
    for(i = 0; i < elimnumber; i++) {                            //forward propagation
        v.b ?+= mvmult(FWD[i], v.b);
    }
    v.x = v.b;
  BACK-SUBSTITUTION:
    for(i = elimnumber-1; i>=0; i--) {                           //back solution
        v.temp = mvmult(BAK[i], v.x);
        v.x ?+= v.ast × v.temp;
    }
    solve = v.x;                                                 //solution vector
}
```

Algorithm 7.3: Parallel Gaussian elimination. The **factor** procedure selects an independent set of vertices for elimination in parallel and modifies the graph structure accordingly. After all vertices are eliminated, a factorization of the system has been computed. The **solve** procedure solves the system for a particular right-hand side.

by the asterate value.

In the serial algorithm, the values of the selected vertices were propagated to their neighbors at this point. In this parallel algorithm, the current set of forward edges at step elimnumber is saved in array FWD so that they may be used in the **solve** procedure. The arrays FWD and BAK record the structure of the graph at each of the elimination steps numbered by elimnumber so that the forward propagation and back solution steps can be performed separately.

Fill-in edge creation could proceed by first creating all pairs of forward and backward edges that meet at selected vertices by using **move-join**. Then, after computing the fill-in values at each site, these could merged with the edges in the *other* vector. However, **mmmult** is used directly by noting that the fill-in values are the product of the matrix partitions fwd and bak. Then, by using **mmadd**, these values can be merged into the matrix represented by the *other* edges. The factorization process repeats until all vertices have been eliminated from the graph and there are no more edges.

The **solve** phase begins with a vector of initial values, b, and performs the forward propagation and back-solve steps using the stored values of the edges at the time of their elimination. As vertices were chosen for elimination at step number i, the scaled forward edges were stored in array location FWD[i]. The forward propagation step involves sending vertex values from the selected vertices over the forward edges and adding them together at their destinations. This is a simple application of **mvmult** as shown earlier. Because the array FWD records the forward edges for each step of the elimination, only those vertices incident to the forward edges at any given step will be modified. Thus, the loop labeled FORWARD in procedure **solve** implements forward propagation for each step in the parallel elimination procedure.

The back-solve phase involves adding vertices back to the graph in the reverse order of their elimination ordering. Once again this requires adding scaled values of the neighbors of the vertices as they are added back. The backward edges stored at array location BAK[i] are used with **mvmult** to add the neighbor values for each of the newly added vertices.

It is interesting to notice that the the edge vectors stored in the arrays FWD and BAK completely describe the vertices selected at each step and the edges incident to them. These vectors are used in the **solve** phase only in the context of a matrix-vector multiplication. If desired, even more work could be moved to the **factor** procedure to speed up **solve** by building the mappings needed for the matrix-vector multiplications during the **factor** stage. When solving the same system for multiple right-hand-sides, this approach would trade off increased factorization time with reduced solution time. The change in the program would involve only exposing the creation of the necessary mappings to the **factor** and **solve** procedures. Note too, that the space required would not be too prohibitive. In exchange for storing the extra mappings, the row and column indices of the elements in the FWD and BAK arrays could be deleted.

7.1.3 Selection of vertices

At each step an independent set of vertices must be selected for elimination. There are many ways to select such a set. For one, the **MIS** procedure could be used. The disadvantage of this approach is that without considering other aspects of the graph's structure, the first few applications of this strategy will create an undue amount of fill-in. The sparsity of the original system will be lost quickly.

Alternatively, if the graph is known to fall into some class for which there are good separators, a variant of the method of parallel nested dissection (which is based on generalized nested dissection) could be used [PR85, LRT79]. This approach calculates an elimination ordering based on the original structure of the graph. A separator set C is found that divides the graph into nearly equal parts, A and B. The separator tree is built recursively by placing the vertices of C in a new tree node whose children are the tree nodes constructed by applying the procedure to A and B. The recursive procedure terminates when each component has just a few vertices. A parallel elimination ordering is then calculated from the leaves up such that no parent is eliminated before any of its children [PR85].

In many supercomputer applications the solution of a system is broken into three stages: ANALYZE, FACTOR, and SOLVE [DR83, ESS81]. The ANALYZE stage performs a symbolic analysis of the structure of the system. It usually calculates an elimination ordering and may optionally allocate space for fill-in values. The numeric factorization stage then performs the actual calculations and the edge values. The parallel algorithm presented here combines the two so that numerical values of the graph can be used for vertex selection. However, procedure **mark** could easily be modified to simply mark those vertices at each step chosen by an earlier ANALYZE stage.

For some applications, numerical stability is an issue. If the system to be solved is known to be positive definite, then vertices can be selected on the structural properties of the graph alone [DR83]. However, for indefinite or unsymmetric systems, the numerical value of the pivots may have to be taken into account [Pis84]. A suitable **mark** procedure could be substituted that attempts to ensure stability.

7.1.4 Complexity

The total number of parallel elimination steps required is a function, $k(n)$, of the structure of the graph and the elimination ordering chosen. For each iteration, the application of **mmmult** dominates the step complexity measure. With this simple fact, the step complexity of **factor** is $S = O(k(n)\lg n)$. The **solve** procedure performs $O(k(n))$ simple matrix-vector multiplications, thus its step complexity is $S = O(k(n)\lg n)$.

This section will use the method of parallel nested dissection to arrive at bounds for the total

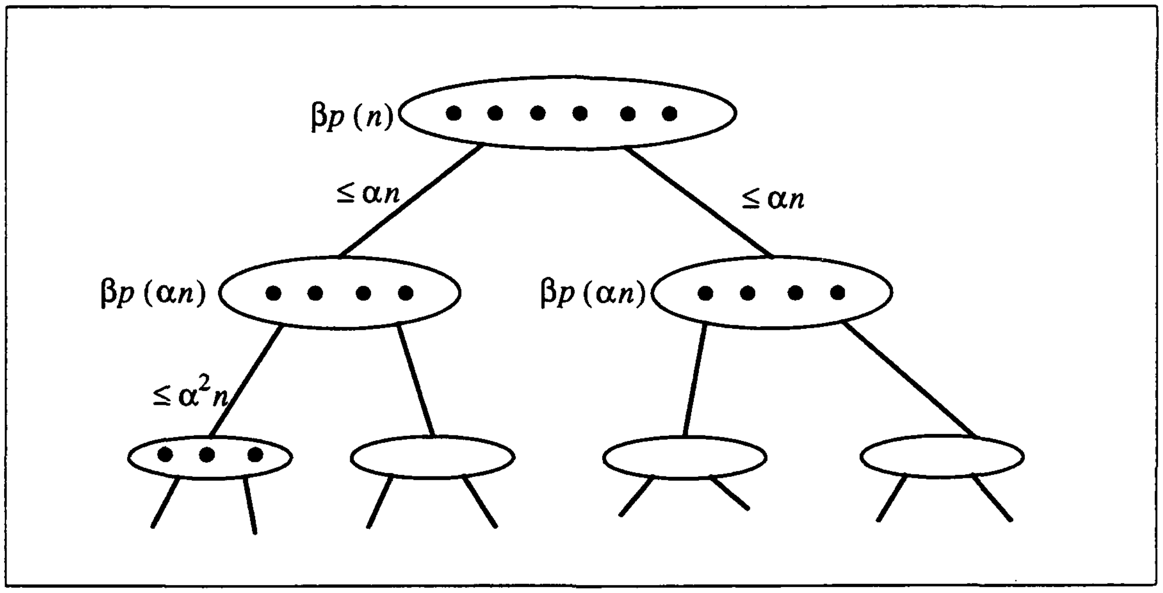


Figure 7.6: An example separator tree. The vertices of the original graph are distributed through the nodes of the separator tree. Edges in the original graph occur only within nodes or along the links of the tree.

step count. A *separator* of a graph is a relatively small set of vertices whose removal causes the graph to fall apart into a number of smaller pieces. If S is a class of graph, and n is the number of vertices, then S satisfies a $p(n)$-separator theorem if there are constants $\alpha < 1$ and $\beta > 0$ such that a separator set with at most $\beta p(n)$ vertices separates the graph into components with at most αn vertices each.

A *separator tree* can be used to describe the decomposition of a graph in terms of its separators. At the highest level is a separator that divides the graph into components. These components themselves have separators, and so on. At the lowest level are components that may be divided no further, possibly containing only a single vertex. An example separator tree is shown in Figure 7.6. The vertices of the original graph are distributed throughout the nodes of the separator tree, and the weights indicated show the sizes of the nodes and subtrees. Because of the separator property, edges in the original graph occur only within nodes of the separator tree, or along the links between adjacent nodes.

Using the separator tree of a graph, a parallel nested dissection ordering is constructed from the leaves up such that no parent is eliminated before any of its children. Because there are no edges between children in different subtrees, one vertex from each leaf node may be eliminated in parallel. Working backwards, the last vertices to be eliminated will be those of the topmost separator, of size $\beta p(n)$. This requires $\beta p(n)$ steps. Preceding these, the vertices of the two subtrees may be eliminated in parallel, and so on. The total step count of the algorithm is described by the following recurrence.

$$k(n) = k(\alpha n) + \beta p(n)$$

Recurrences of this form occur frequently in the analysis of divide-and-conquer algorithms. Assuming that $p(n) = n^\sigma$, the solution to this recurrence is given by the following formula, adapted from [LD91].

$$k(n) = \begin{cases} O(\log n) & \text{if } \sigma = 0 \\ O(n^\sigma) & \text{otherwise} \end{cases}$$

These formulas for the elimination step count will now be applied to certain classes of graphs.

Binary trees are a class of graph that is closed under subgraph; separation at any vertex separates the graph into two smaller binary trees. The following theorem is stated for binary trees [Gil80].

Theorem 1 *The class of binary trees sati'.es a 1-separator theorem for $\alpha = \frac{2}{3}$ and $\beta = 1$.*

Series-parallel graphs also have constant sized separators. These graphs obey the following theorem.

Theorem 2 *The class of series-parallel graphs satisfies a 2-separator theorem for $\alpha = \frac{2}{3}$ and $\beta = 1$.*

Because $p(n) = n^0 = 1$ for both of these classes of graph, the total step complexity of the parallel Gaussian elimination **factor** procedure would be

$$S = O(\log^2 n).$$

Many graphs that occur in digital logic networks are nearly series-parallel and have small, constant sized separators. These bounds could be met for finding transitive closure in those systems too.

A planar graph is one which can be drawn on a plane so that the edges of the graph only intersect at their endpoints [BM82]. Finite element meshes are one example of important applications that are described by planar graphs. The following theorem is due to Lipton and Tarjan [LT80].

Theorem 3 *The class of planar graphs satisfies a $\sqrt{n}$-separator theorem for $\alpha = \frac{2}{3}$ and $\beta = 2\sqrt{2}$.*

Thus, $p(n) = n^{1/2}$, and the total step complexity of the Gaussian elimination procedure is

$$S = O(\sqrt{n} \log n).$$

The total amount of work performed by the factorization algorithm is dominated by the number of partial products computed by the **mmmult** function. If every elimination step resulted in massive (or nearly complete) fill-in, then the total work would be $W = O(Sn^3)$, where S is the step complexity of the **factor** procedure. However, a generalized nested dissection ordering based on graph separators limits the total amount of fill-in. The separator tree may be used to analyze the total amount of fill-in that a parallel elimination ordering produces. A theorem by Rose and Tarjan states that an edge (v, w) fills in if and only if there is a path from v to w containing only vertices eliminated before either v or w [RT78]. Relating this to our separator tree framework, the separator tree shows that the only possible fill-in that may occur is along the edges of the tree, or between the vertices of an individual node of the tree. Analysis of the work complexity of the algorithm proceeds by computing the total number of fill-in values produced at each elimination step, based on the structure of the separator tree.

Such a detailed analysis is beyond the scope of this document, but by using some of the results of Lipton, Rose and Tarjan bounds may be placed on the total number of partial products computed by relating it to the multiplication count of the serial algorithm. For planar graphs, the method of generalized nested dissection ensures that the total multiplication count of the algorithm is $O(n^{3/2})$ [LRT79]. Because this is same as the number of partial products calculated, for the planar graphs the work complexity of our **factor** algorithm is

$$W = O(Sn^{3/2}) = O(n^2 \log n).$$

This is a substantial improvement over the pessimistic bounds given above. For graphs with constant sized separators, such as binary trees and series-parallel graphs, the multiplication count is $O(n)$. For these, the work complexity of our algorithm is

$$W = O(n \log^2 n).$$

7.1.5 Systems of Linear Equations

The Gaussian elimination algorithm described is generalized for the solution of systems of equations over any semiring. When the system involves linear equations, the mathematical operations may be simplified somewhat. The algorithm described solves a system of the form

$$Ax + b = x.$$

This form is slightly different than the traditional form that systems of linear equations appear in which is generally

$$A'x = b.$$

Thus, when using the above procedure on linear systems, the actual edge labels should come from the matrix $A = I - A'$.

However, the matrix A' can be used directly with a simple modification to the algorithm. First, the asteration operator is changed to be

$$a^* = -\frac{1}{a}.$$

With this simple change, the matrix A' may be used directly to label the edges, and the Gaussian elimination algorithm above will compute $(-x)$. Thus, the final labels of the vertices will be the negative values of the desired solution.

Digression

The mathematical basis for this change is not too difficult to understand. The set of all systems of equations over a semiring are themselves a semiring [AS85]. With the definition of asteration given earlier, $a^* = aa^* + 1$, for a system of equations over the semiring the corresponding matrix operation is

$$A^* = AA^* + I = A^*A + I.$$

For such a definition of the elements of the matrix, the Gaussian elimination algorithm correctly calculates A^*b. By substitution, this is shown to be the solution by using simple algebra and the definition of asteration on the system of equations.

$$\begin{aligned} Ax + b &= x \\ A(A^*b) + b &= x \\ (AA^* + I)b &= x \\ A^*b &= x; \end{aligned}$$

For linear equations of the form

$$Ax = b,$$

with the definition of asteration changed to $a^* = -1/a$, the corresponding implicit solution to matrix asteration is

$$AA^* + I = 0.$$

For linear systems, this is more typically represented as

$$A^* = -1/A.$$

Note that these expressions involve both additive and multiplicative inverses, so this is no longer a semiring. However, it is the familiar realm of linear systems.

Asteration of a matrix is defined inductively [Leh77]. For a one element matrix, $[a]^* = [a^*]$. For a larger matrix

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

its closure is

$$A^* = \begin{bmatrix} B^* + B^*C\Delta^*DB^* & B^*C\Delta^* \\ \Delta^*DB^* & \Delta^* \end{bmatrix} \text{ where } \Delta = E + DB^*C$$

With the new definition of asteration for linear systems, it is shown that $AA^* + I = 0$ by showing that $AA^* = -I$.

First, for the base case

$$aa^* = a\frac{-1}{a} = -1.$$

$$\Delta^* = \frac{-B}{BD - CD}$$

$$A^* = \begin{bmatrix} \frac{-1}{B} - \frac{CD}{B(BE-CD)} & \frac{C}{BE-CD} \\ \frac{D}{BE-CD} & \frac{-B}{BE-CD} \end{bmatrix}$$

$$A^*A = \begin{bmatrix} (-I - \frac{CD}{BE-CD} + \frac{CD}{BE-CD} & \frac{BC}{BE-CD} + \frac{-BC}{BE-CD} \\ \frac{BD}{BE-CD} + \frac{-BD}{BE-CD} & \frac{CD}{BE-CD} + \frac{-BE}{BE-CD} \end{bmatrix}$$

$$A^*A = \begin{bmatrix} (-I - \frac{CD}{BE-CD} + \frac{CD}{BE-CD} & \frac{BC}{BE-CD} + \frac{-BC}{BE-CD} \\ \frac{BD}{BE-CD} + \frac{-BD}{BE-CD} & \frac{CD}{BE-CD} + \frac{-BE}{BE-CD} \end{bmatrix} = \begin{bmatrix} -I & 0 \\ 0 & -I \end{bmatrix}$$

Our parallel Gaussian elimination algorithm calculates A^*b, but for linear equations of this *form, the result calculated is the negative of the real solution.*

$$\begin{aligned} Ax &= b \\ A^*Ax &= A^*b \\ A^*Ax + x &= A^*b + x \\ 0 &= A^*b + x \end{aligned}$$

Thus, with this change to the definition of asteration, our algorithm finds the additive inverse (negatives) of the values of the solution.

7.1.6 Relationship to LU factorization and block methods

The procedure described above corresponds directly to LU factorization. The array FWD describes the U matrix and BAK describes the L matrix. The **solve** procedure solves the simplified systems $Ly = b$ and $Ux = y$ The sets of parallel variables eliminated also describe a blocking of the variables into sets such that groups of rows participate in each forward or back substitution step.

The process is somewhat analogous to what is known as the multifrontal method for block factorization [ADD89, DR83], but a more similar use of the parallel elimination ordering we presented was developed in the distributed multifrontal method of [LBT87]. At each step a number of vertices are selected for elimination. With a suitable reordering of the variables, this step corresponds to a partitioning of the matrix into parts

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

The relationship between this partitioning and ours is simply B=BAK, C=FWD, D=OTHER and A=SELF. Because we have restricted vertex selection to be an independent set, A is a diagonal matrix. However, in the general multifrontal method, A may not be diagonal. The multifrontal method updates the lower right partition (OTHER edges) by computing its "Schur complement,"

$$D - CA^{-1}B.$$

Since our A matrix is a diagonal, its inverse is trivial and the scaling of the forward edges represents the product $(A^{-1}B)$. The matrix product and matrix addition of our algorithm implement the remainder of the computation of the Schur complement.

Part III

Implementation

Implementation

The following chapters describe the implementation of the **match** and **move** operators in detail for two different parallel computing models. One is based on EREW memory access and a sorting procedure, the other on CRCW memory access and parallel hashing. Both of these models are supported to varying degrees on each of the Connection MAchine CM-2 and the CRAY Y-MP. However, full support of the CRCW model on the CRAY Y-MP requires development of the multprefix operator presented in Chapter 10.

Chapter 11 compares actual performance figures for each of these approaches on each of the two machines. The data collected show that our high-level approach to supercomputing using **match** and **move** is competetive with machine-specific code written for each of the architectures examined.

Chapter 8

Implementing match using sorting

This chapter describes the implementation of the **match** and **move** using sorting and the segmented-**scan** operations of the strict scan vector model. In this model, communication primitives **send** and **get** only permute values; **send** does not combine values and **get** does not support concurrent reads. This is the parallel vector analog of the EREW PRAM model.

The implementation strategy brings equal keys near one another by sorting, and segmented-**scan** operations serve as basic mechanism for replicating or combining values. Before discussing the implementation in detail however, some more general concepts about mappings are presented. The next section describes the type of information that a mapping must supply so that **move** may be implemented efficiently. Then "group numbers" are described, which are used in the description of the **match** and **move** algorithms that follow. The concepts presented are used in this chapter, and also the next which describes an implementation of **match** that uses parallel hashing instead of sorting.

8.1 Mappings

A mapping is a data structure that describes the paths indicated by a **match** operation. As explained earlier, **match** may also be viewed as collecting the source and destination sites into groups. The main task of **match** is to identify the distinct groups and to calculate information about them. The information calculated summarizes some of properties of the different groups. These include:

- The number of source sites for each group.
- The number of destination sites for each group.
- Source and destination sites that did not match anything.
- Locations of the source and destination sites for each group.
- For each group, the location where some sort of group combining operation should occur.

Whether implementing **match** using sorting, as described in this chapter, or by the parallel hashing method of the next chapter, these pieces of information are necessary for the efficient implementation of the **move** operations.

8.2 Major and Minor group numbers

The description of the **match** and **move** operations make frequent use of the concept of a group. One of the important things that **match** does is to perform a "canonicalization" step that renames each group from an arbitrary key value to a small integer value. For each source or destination site in a mapping, this number is its "major" group number. In a **match** that results in G groups, the major group numbers lie in the range $[0..G-1]$.

Each key site also has an associated "minor" group number. For a group with S source keys, these values lie in the range $[0..S-1]$ and arbitrarily enumerates the source keys of the group. Similarly, the D destination keys are enumerated in the range $[0..D-1]$.

Major and minor group numbers are a convenient way to differentiate groups, and to differentiate sources and destinations within groups. Minor group numbers will become important later in the implementation of **move-join**.

8.3 Matching by Sorting

The idea behind constructing a mapping through sorting is that equal keys should be brought near each other by a sorting operation. This type of mapping associates the groups specified by keys with a segment in a vector. The combining function appropriate for this type of group is the segmented-**scan**.

The steps involved in building a mapping are illustrated in Figure 8.1. In our example source keys are shown in uppercase while their corresponding destination keys are shown in lower case. The subscripts are the minor group number of each key and serve to differentiate the members of the same group in our examples. (Minor group numbers are calculated later in the implementation of the **move-join** function.)

First, the keys of the source and destination vector are concatenated into one vector called the "agent" along with pointers back to their sites of origin. These site addresses are called the

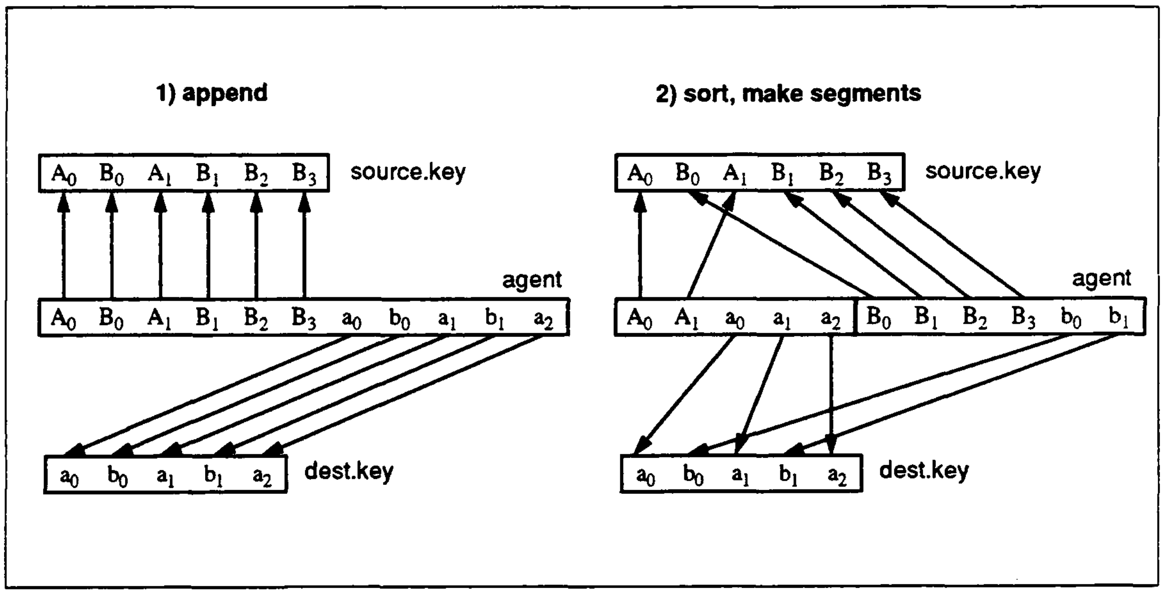

1) append
2) sort, make segments
source.key
agent
dest.key
A_0 B_0 A_1 B_1 B_2 B_3
A_0 B_0 A_1 B_1 B_2 B_3 a_0 b_0 a_1 b_1 a_2
a_0 b_0 a_1 b_1 a_2
A_0 A_1 a_0 a_1 a_2 B_0 B_1 B_2 B_3 b_0 b_1

Figure 8.1: A mapping may be created by using sorting. In the first step, keys are appended from the source and destination vectors into a new vector, called the "agent" vector. After sorting the agent sites, they are divided into groups by comparing each key to its neighbors.

"home" pointers for the keys. After sorting the agent elements by their key value, the agents are divided into segments so that each group of equal keys is in one segment. The identification of segments is done in one parallel step by comparing each key to its left neighbor in the sorted agent vector. Finally, the indices of the new sites in the agent vector are sent back to the home site of each key. The agent vector along with the pointers from keys to agent sites becomes the data structure of the mapping.

A **move** operation using this type of mapping is illustrated in Figure 8.2. Three steps complete the **move**. First, the source values are sent to the agent. Next, a single **scan** combines the values of the source agents and distributes the sum across the destination elements. The last step sends the sums in the destination agents to their final sites

Because the agent sites are arranged with source agents before destination agents, a single segmented **scan** combines the source values and distributes them across the destination agents. By ensuring that the destination agents are initially set to 0 (or an appropriate identity element for the **scan**), a single scan serves the purpose of both combining the data from the sources, and replicating it for all destinations.

In presenting this simple overview, many important details were omitted. **null** keys must be accounted for, as must be destination sites that match no source sites. The numbers of source and destination keys belonging to each group must also be computed. The next section considers the implementation of **match** in more detail.

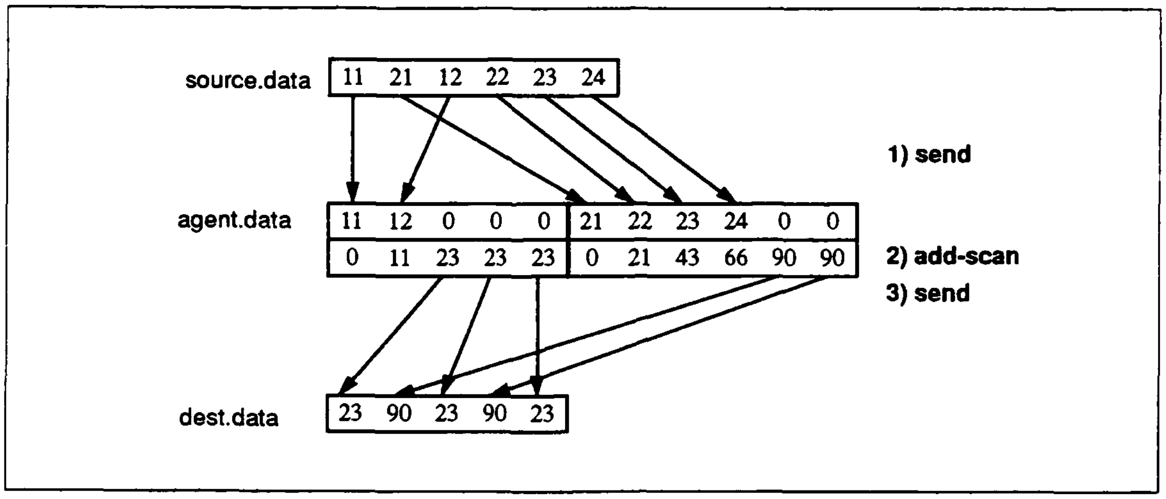


Figure 8.2: Using a mapping created by sorting to perform a **move-add**. The first step sends the source values to the agent vector, where they are combined with a segmented **add-scan**. The final **send** forwards the results to their destinations.

8.4 match

The sorting step arranges equal keys into segments. Then, segmented scan and reduction operations may calculate information about each group. As shown, the segments created in this process are also used to perform combining for the basic **move** operations. The mapping created is a data structure that stores information about the source and destination key sites, the segments of the agent, and other information about the groups. It is a data structure that spans many vector sets, adding fields to each of them. This section describes the fields of the mapping that are created by **match**.

Code for the **match** operation is shown in Algorithm 8.1. The first action it performs is to add fields called active to each of the source and destination vector sets. This Boolean field is **true** for each source site that matches a destination. Similarly, a destination key is active if it matches a source. Because a **null** key cannot match anything, these fields are initialized to **true** only for non-null keys. Later on, additional active sites may be deactivated if it is discovered that they match no other.

The active keys are appended to construct the vector set called the "agent." The agent vector set has one site for each key participating in the **match**. Function **cond-append** is used to append only those keys that are active; its implementation is omitted because it is trivial. The information sent to the agent vector set includes the source and destination keys (field key), a flag value indicating whether the key is a destination key or not (dest), and the site index of the origin of the key (home). Note that with the dest flag and the home value, the home site of every key is uniquely identified.

The keys are sorted in a two-step process. Function **rank** produces a permutation vector

```
match(dest.key, source.key) {
    source.active = (source.key ! = null);
    dest.active = (dest.key ! = null);
    agent.<key,dest,home> = cond-append (source.<key,0,#>, dest.<key,1,#>,
            source.active, dest.active);

    //Rank source agents before dest agents
    rank(agent.rank, agent.<key,dest>);
    send(agent.<key,dest,home>, agent.rank, agent.<key,dest,home>);

    //segments by key
    asd.<len,start> = make-seg-descr(agent.key);

    //Count destination elts per group
    seg-add-reduce(asd.dcnt, agent.dest, asd.len);

    //Calc number of source sites per group
    asd.scnt = asd.len - asd.dcnt;
    //Distribute this information over the agent sites
    seg-distr(agent.<scnt,dcnt>, asd.<scnt,dcnt>, asd.len);

    //Calculate whether agent sites are active, then send home
    agent.active = agent.dest ? (agent.scnt ! = 0) : (agent.dcnt ! = 0);
    dest.agentactive = 0;                        //copy of active flag from agent site
    cond-send(dest.agentactive, agent.home, agent.active, agent.dest);
    dest.active &= dest.agentactive;
    source.agentactive = 0;
    cond-send(source.<active,asite>, agent.home, agent.<active,#>, ! agent.dest);
    source.active &= source.agentactive;

    //Construct the mapping
    M = MAPPING(source.<asite,active>, dest.<active>,
            agent.<dest,home>, asd.<scnt,dcnt,len,start>);
}
```

Algorithm 8.1: Implementing the **match** operation in the EREW scan vector model. With the exception of the **rank** subroutine, all other instructions are unit step operations.

```
make-seg-descr (a.key) {
    shift-right(a.lkey, a.key);
    a.startbit = (a.lkey != a.key);              //start if keys differ
    a.startbit = (a.# == 0) ? 1 : a.startbit;    //first site always starts a seg

    //Enumerate startbits: implement a pack
    newlength = add-scan(a.newpos, a.startbit);
    asd = new(newlength);                        //agent seg descriptor vect
    cond-send(asd.start, a.#, a.newpos, a.startbit);

    //Calculate length of segments
    shift-right(asd.rstart, asd.start);
    asd.len = asd.start - asd.rstart;            //subtract start to left
    asd.rstart = (asd.# == 0) ? 0 : asd.rstart;  //fix first site
    make-seg-descr = asd.<len,start>;
}
```

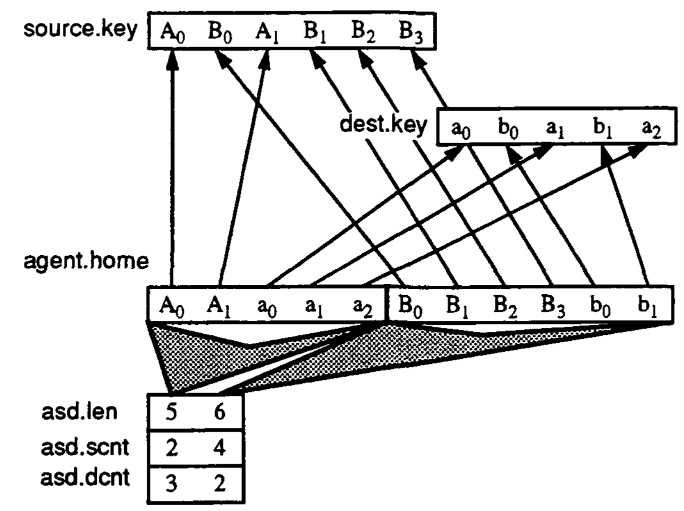
Algorithm 8.2: Computing a segment descriptor from the sorted keys. A new segment begins where a key differs from the one to its left.

that may be used with **send** to put the keys in sorted order. The keys are actually sorted by a composite value: <key,dest>. This ensures that after they are sorted, for each group of equal keys, destination keys follow source keys. If the sort is stable, the dest flag would not need to be included in the sort-key.

At this point, the keys are in sorted order, with source keys preceding destination keys. The next important step identifies segments so that equal keys are in the same segment. Function **make-seg-descr** performs this operation; its implementation is shown in Algorithm 8.2. A new segment is identified wherever a key differs from the one to its left. To perform the comparison, the key field is shifted right, and then in an elementwise fashion, the comparison is performed. This produces a start-bits type of segment descriptor, a.startbit, from which a segment-length type must be computed. A few vector operations suffice to perform this transformation.

The vector set that describes the segmenting of the agent is called asd, an abbreviation for "agent-segment-descriptor." The len field gives the length of each segment, while the start field gives the starting site of each corresponding segment. The start values are often used to send a value to the front of each segment so that it can be copied across using a **copy-scan**.

The segment descriptor vector set has one site for each group involved in the **match**. Because there is one site in this vector set, its index, asd.#, assigns the major group numbers. The next important information calculated are summary data about the groups. First, the number of destination keys in each group is calculated by adding up the dest flags in each segment using **seg-add-reduce**. A count of the number of source keys can be calculated in an elementwise fashion by subtracting this value from the total length of each segment. These values are called



source.key
A_0 B_0 A_1 B_1 B_2 B_3
dest.key
a_0 b_0 a_1 b_1 a_2
agent.home
A_0 A_1 a_0 a_1 a_2 B_0 B_1 B_2 B_3 b_0 b_1
asd.len
5 6
asd.scnt
2 4
asd.dcnt
3 2

Figure 8.3: Relationship between source, destination, agent and segment descriptor vector sets. Each group of keys is arranged in one segment. The count of the number source and destination keys in each group is an important quantity computed by **match**.

scnt and dcnt. Figure 8.3 illustrates these relationships between the source, destination, agent and segment descriptor vector sets, and shows the important values calculated.

The scnt and dcnt values are distributed back over the segments for each group so that each agent site can determine if it is active or not. This information will be sent back to the home site of each key. In an elementwise statement, a destination type agent site remains active if its group has a non-zero scnt value, indicating that a destination site matches at least one source. Similarly, a source type agent site is active if its group has a non-zero dcnt value.

The final two steps forward information about the agent sites back home to the source and destination vector sets. For the destination, only the active flag needs to be sent. The effect of this statement is that in addition to marking **null** keys as inactive, those that matched no source also become inactive. This information is used to ensure that inactive destination sites receive the **null** value as the result of a simple **move** operation.

For the source, the same active flag information is sent home, as is the position of the key in the sorted agent vector. This value is stored in the asite ("agent-site") field of the source. When implementing a combining **move**, inactive source sites need not send their values to the agent vector because they are not used.

The final mapping data structure is created in the last step. It is a collection of fields from the source, destination, agent, and segment-descriptor vector sets as listed. Other fields created during the implementation of **match** can be freed.

8.4.1 Complexity

All operations in the implementation of **match** are primitives in the scan vector model except for the **rank** operation. Thus, the complexity of **match** is completely dictated by the complexity of the sorting operation used in **rank**. In general, sorting requires $S = O(\lg n)$ steps and $W = O(n \lg n)$ work in the scan vector model. Section 4.1 discussed the complexity of sorting in this model and showed some of the types of optimizations that can be performed in special cases.

8.4.2 Multi-field match

match is defined so that it can match composite keys that are tuples as well as the simple atomic types. The simplest way of implementing a multi-field **match** is to pack the fields into a single machine word that is then used as the key. The packing step is performed elementwise and adds very little overhead to the procedure.

When the fields of the **match** cannot be packed into a single machine word, multiple words must be moved explicitly in each of the **send** operations, and the ranking operation must be performed on a multi-word key. For a w-word key this adds $S = O(w)$ steps to the **match** algorithm. However, since w will be constant for a given algorithm, only a constant number of steps is added to an algorithm using a multi-word **match**. In practice, most algorithms can be handled by the single word packing approach.

8.5 move

The basic combining **move** operations are all implemented using the same procedure: the only thing that changes is the type of scan used for combining and the value of the identity element (0) for the combining function chosen. The complete **move** procedure is shown in Algorithm 8.3. Recall that the task of a combining move is to combine values from the active source sites and forward them to the matching destination sites. Destination sites that are not active should get the **null** value.

In the first step, a data field is constructed in the agent called agent.data to which the source data from the active sites are sent. Its sites are initialized to the appropriate identity value for the combining function chosen. For a **move-add** this is 0. For a **move-max** the identity element should be changed to $-\infty$, etc.

Because source agents precede destination agents in each segment, a single segmented-scan suffices to combine all of the source values and copy them for each destination. Destination agent sites, which have been initialized to the appropriate 0 value, obtain the final sum of the source values. The result values are sent to their destination, d.data, using a conditional send. Lastly, inactive destination sites are set to **null**.

```
move-op(M, source.data) {
    //Initialize data to appropriate zero value for 'op'
    agent.data = 0;

    //Send to agent site only if active
    cond-send(agent.data, source.asite, source.data, source.active);

    //Scan to combine sources, replicate for destinations
    seg-scan-op(agent.data, agent.data, asd.len);

    //Forward to dest
    cond-send(dest.data, agent.data, agent.home, agent.dest & agent.active);

    //Set non-active dest sites to null
    dest.data = dest.active ? dest.data : null;
}
```

Algorithm 8.3: Implementing the **move** operations in the EREW scan vector model. All instructions are unit step in the scan vector model.

8.5.1 Complexity

All of the operations implementing a combining **move** are primitives in the scan vector model. Thus, the algorithm's step complexity is $S = O(1)$. The work complexity of the algorithm is the sum of the lengths of the source and destination vector sets: $W = O(s + d)$.

8.5.2 multi-field moves

All of the **move** variants are defined to operate on tuples of values as well as simple atomic values. In many instances it is simple to pack a number of fields together into one machine word before the **move** is executed, and then unpack the fields at the result. For instance, in many of the graph and sparse matrix algorithms, the fields moved are pairs of integers that may be packed into a 32 or 64-bit word. Because the packing and unpacking steps are elementwise operations, only a constant number of steps is added to an algorithm that uses a combining **move** on tuple values.

When the moved tuples do not fit in a single word, a multi-word procedure must be used. The two **send** operations are simple to extend to multi-word operation, but the combining requires simulating a multi-word scan operation from simple scan operations. Procedures for performing this simulation are outlined only briefly. Each of these procedures requires $S = O(w)$ steps where w is the number of machine words combined

For **move-min** and **move-max**, the procedure begins at the most-significant word and proceeds downward. The procedure is shown in Algorithm 8.4 for a **move-max**. The selection of the current

```
agent.enabled = 1;
for (i = (w-1) downto 0) {
    seg-reduce(asd.max[i], asd.len, agent.enabled ? agent.val[i] : $-\infty$);
    seg-distr(agent.max[i], asd.len, seg.max[i]);
    agent.enabled &= (agent.max[i] == agent.val[i]);
}
```

Algorithm 8.4: Simulating multi-word combining for a **move-max** function using primitive single-word **scan** operations. For a w-word combining **move**, this procedure adds w steps to the algorithm.

word is indicated as an array reference into the fields. All agent sites are initially marked as "enabled" for this procedure and remove themselves from consideration when their source value is not chosen as the maximum. A site disables itself by substituting the appropriate identity value for the combining function. For each word, the maximum value of each segment is collected in asd.max. This value is then distributed back over the agent segments. Any agent site that does not have an equal value determines that it is not the maximum and disables itself.

The **move-arb** function may be implemented by either **move-min** or **move-max**. The only other multi-field combining **move** is **move-add**. A multi-field **move-add** implies that overflow information for an addition on each field is carried over into the next significant field. Such a capability would require that a machine be able to calculate overflow beyond what can be represented by a single machine word. This is an unreasonable assumption for most machines. For this reason, a multi-field **move-add** is implemented to add each of the fields separately.

8.6 Optimizations

Section 4.1 considered many optimizations that can be employed for special situations using **match**. These all affected the asymptotic complexity of algorithms written using **match**. In this section we discuss some optimizations that can reduce the constants associated with an implementation of the operator.

The fields of the mapping data structure created for these special cases may be slightly different, requiring modified **move** operations. However, these differences are hidden from the programmer using the the **match** and **move** operations. Thus, a programmer using **match** and **move** may write programs for one unified model, but is assured that suitable variants of the algorithms are used to obtain high performance.

8.6.1 "NoNulls" fields

When the source and destination fields of a **match** operation both have the NoNulls property, the implementation may choose to avoid tests for **null** key values. The **conditional-append** may be foregone, and a regular vector **append** can be used. On some machines, (such as the CRAY Y-MP) the unconditional append is much faster as it avoids address arithmetic, and becomes two simple block copy instructions. Algorithms on sparse matrices in particular involve no **null** valued keys.

8.6.2 Self-match

A self-match is a **match** operation where the source and destination are the same field. In this case, no **append** function needs to be performed at all. The keys are merely sorted and half of the other fields can be omitted, since they are the same for the source and destination. Because only half as many keys are sorted as there would be with the regular method, the ranking step is twice as fast. (Note that if the key field is marked with the "Sorted" property, then no ranking needs to be performed.)

Because the agent vector has no destination sites, the **move** procedure will be slightly modified. Rather than using a single **seg-scan** to combine and replicate the values, a two-step process is used. First, a **seg-reduce** is used to add up the source values, and then a **seg-distr** is used to replicate them across the agent to send them back to the source sites. The fields in this type of mapping are illustrated in Figure 8.4. The changes to the **move** procedure for this special type of mapping are trivial.

8.6.3 Library Function collapse

The **collapse** operation creates one site in a new vector for each distinct key. It can be implemented using the **match** and **move** primitives, but because of its frequency of use, **collapse** should be implemented at a low level. Its implementation is very similar to that of a self-match.

A representation of a special collapse-mapping is shown in Figure 8.5. After sorting the keys of the single argument key vector, they are segmented as before. The segment descriptor vector set, asd, has one site for each distinct key and is the destination vector set of the **collapse** mapping. A **move** operation performed on such a mapping sends values to the agent and combines the values for each group using a segmented-reduce function. The reduction vector is the desired result.

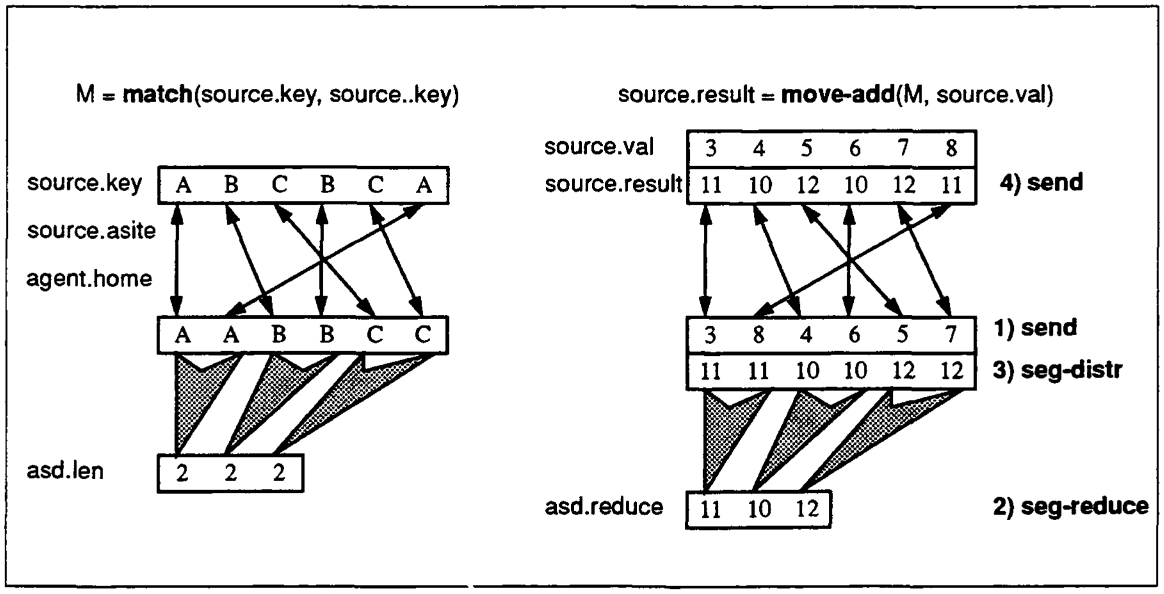


Figure 8.4: A Self-Match mapping. When the source and destination of the **match** are the same field, a slightly different approach is used. The source keys are sorted and segmented as before. However, for the combining **move** operations, a two-step process is used to first combine, and then replicate the values.

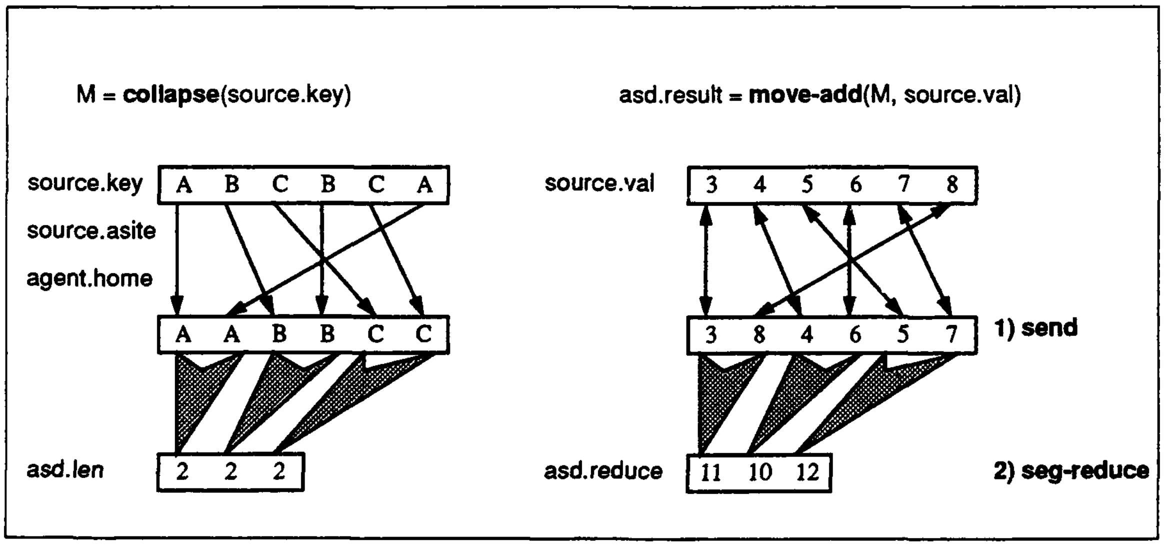


Figure 8.5: A **collapse** mapping. This type of mapping is closely related to the self-match mapping, but only two steps are needed to implement a combining **move**. Because of the frequency of the use of **collapse**, support for this mapping as a special type can yield a significant performance enhancement for many algorithms.

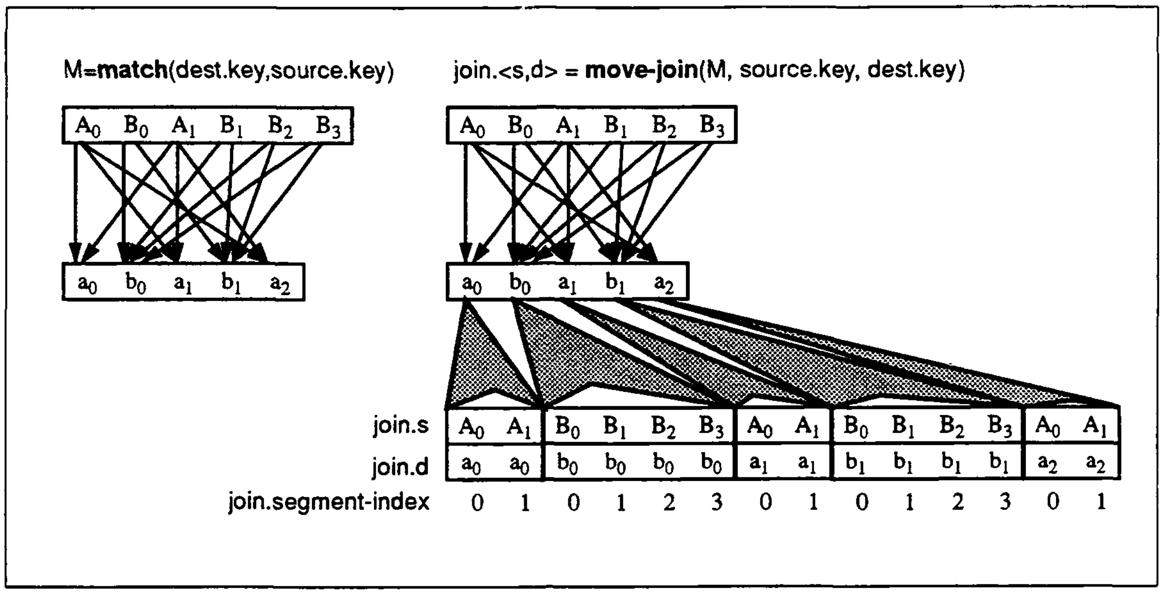

M=match(dest.key,source.key)
join.<s,d> = move-join(M, source.key, dest.key)
A0 B0 A1 B1 B2 B3
a0 b0 a1 b1 a2
join.s
join.d
join.segment-index
A0 A1 B0 B1 B2 B3 A0 A1 B0 B1 B2 B3 A0 A1
a0 a0 b0 b0 b0 b0 a1 a1 b1 b1 b1 b1 a2 a2
0 1 0 1 2 3 0 1 0 1 2 3 0 1

Figure 8.6: Arrangement of the join results with respect to the source and destination key vectors. The values from the source vector are arranged so that the minor group numbers of their sites of origin match the segment index of their final position.

8.7 move-join

The **move-join** operation is substantially different from the other **move** operations in its functionality and implementation. At first glance it seems to not be related to the other **move** operations at all. However, there are a number of reasons why it is very similar to the other **move** operations. First, all of the **move** operations have an $S = O(1)$ step complexity. While the simple combining **move** operations have a work complexity of $W = O(s + d)$, the work complexity of **move-join** is $W = O(s \times d)$. Their step complexity differs from **match** which is $S = O(\lg n)$. Secondly, **move-join** rearranges data, which also relates it to the simple combining **move** operations. Finally, **move-join** can be viewed as a combining move if the combining operator is considered a vector concatenation function: in the scan vector model the source values arriving at one destination are combined to fill contiguous sites in a segment in a new vector.

move-join creates a new site for each pair of source and destination site that match. For a group with scnt source sites and dcnt destination sites, this requires creating (scnt×dcnt) sites. When illustrating the **move-join** operation, it is shown with the result sites fanning out from the destination vector, producing one new site for each matching source. For each destination site, the fanned out sites form a segment whose length is the number of source sites in its group. If the sites of each segment are enumerated, then the source values copied to these sites are those of the same group with their minor group number matching the index of the site within the segment. Figure 8.6 illustrates this arrangement.

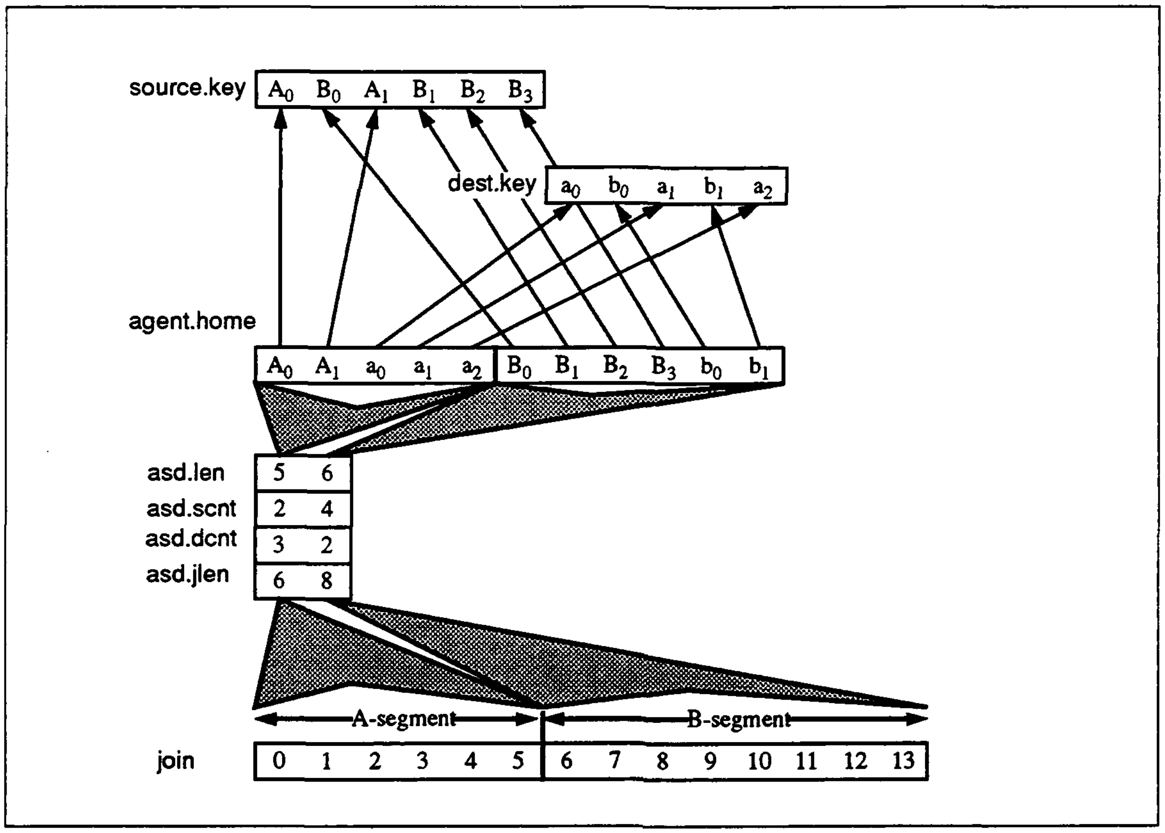

source.key
A_0 B_0 A_1 B_1 B_2 B_3
dest.key
a_0 b_0 a_1 b_1 a_2
agent.home
A_0 A_1 a_0 a_1 a_2 B_0 B_1 B_2 B_3 b_0 b_1
asd.len 5 6
asd.scnt 2 4
asd.dcnt 3 2
asd.jlen 6 8
A-segment
B-segment
join
0 1 2 3 4 5 6 7 8 9 10 11 12 13

Figure 8.7: The number of join sites for each group is computed as the product of the number and source and destination keys. These quantities were computed for each group during the **match** operation.

8.7.1 Implementing move-join

In the segment descriptor vector set, asd, the numbers of source and destination sites in each group are stored in fields asd.scnt and asd.dcnt. The first step of the **move-join** procedure calculates the number of join sites required for each group in field jlen as the product of the number of its source and destination sites. For each key a "join segment" is allocated with jlen sites. The field asd.jlen becomes a segment descriptor for the join vector set. A field called jstart gives the start position of each segment in this vector.

Figure 8.7 illustrates the complicated relationships between the source, destination, agent, segment descriptor and join vector sets. This arrangement is unusual in that the segment descriptor vector set describes the segmenting of two other vector sets through fields len and jlen. Creating the necessary join sites is easy: it is the difficult job of the **move-join** operator to move all of the data to the correct place.

The strategy we will use will be to first move data from the source and destination sites to the join segment of their group. There, it will be replicated as necessary using **copy-scan** operations. Finally, it will be permuted to arrange it in the positions required. However, rather than moving actual join data around, the computation will be performed on site addresses so that the join is

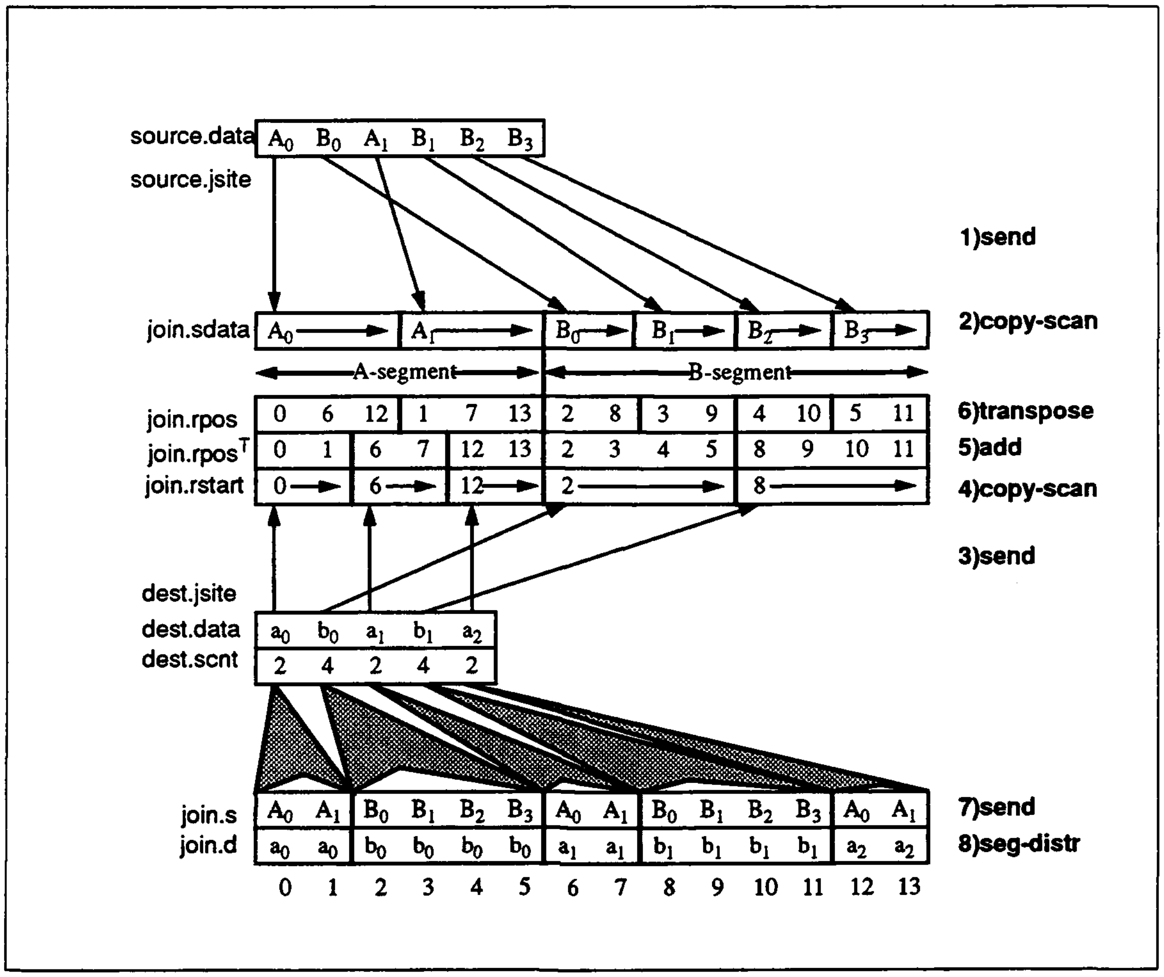


Figure 8.8: The steps of the **move-join** procedure. Segmented **distribute** and **copy-scan** instructions are used to make the necessary copies of data values, and strict permutation instructions send the copies to the correct places.

implemented with a symbolic portion and a following step that actually moves data fields. This is in contrast to a similar algorithm presented in [KRS89] that computes cross-products by groups.

Figure 8.8 illustrates the implementation of the **move-join** operation for the same two key vectors used earlier. Source keys are shown in uppercase, while their matching keys are shown in lowercase. The subscript of each key is its minor group number, and is shown only to differentiate the keys of the same group. The vector operations performed are shown on the right hand side with step numbers showing their sequencing. In the first step, source values are sent to their join segment with a spacing factor equal to the number of destination sites that each matches. After the following **copy-scan**, there are enough copies of each source value for the join result. However, the copies must be rearranged. The next steps are concerned with computing a permutation vector that will forward these copies to the correct result site.

In the final result, each destination site fans out to a number of join sites. The start address of these segments is calculated by performing an **add-scan** on dest.scnt, which is the segment-length descriptor for the join result. Step 3 sends these start address to sites in the field join.rstart where it is copied across in a manner similar to the way source values were copied. The segmenting of join.rstart is described by the site addresses referenced by dest.jsite. After enumerating the sites within each of these segments, this offset is added to join.rstart to obtain join.rpos^T, which is a permutation vector.

The key step of the **move-join** procedure is the transposition performed in Step 6. Notice that the join segments for keys A and B are themselves segmented in two different manners. For key A, there are 2 source keys, and 3 destination keys. Its join segment may be viewed alternatively as 2 segments of length 3, or as 3 segments of length 2. The transposition function permutes the sites of each join segment to relate these two segmentations of the same vector.

Transposition rearranges the sites of each join segment to compute join.rpos, a field that permutes the sites of join.sdata to their final position in join.s, as performed in Step 7. Notice how the permutation sites of the copies of A_0 in join.sdata are sites 0, 6 and 12 in the result vector, as required. This completes the data movement associated with values from the source vector set. Step 8 merely copies the values of the destination sites across the segments described by dest.scnt. The result field has one site for each pair of source and destination for each group, as desired.

The full procedure that implements **move-join** appears in Algorithm 8.5. It performs all the computations described in the preceding paragraphs but also deals with some of the details omitted from the discussion for clarity. Some of these fields calculated include the segmented-index values and the segment start-bit descriptors shown for the fields join.sdata and join.rstart. The implementation of the transposition function is given in Algorithm 8.6 and is relatively straightforward.

move-join is conceptually implemented in two phases: a symbolic address arithmetic portion, and a later portion that moves the actual data. After the symbolic portion, the permutation vector and segment descriptors are added to the mapping data structure so they may be reused. In this manner, when moving many fields with a **move-join**, the symbolic address arithmetic is only performed once.

8.7.2 Further Details

Of course, all of the details are given in the code, but some of the important concepts are discussed here. First of all, the minor group numbers of each key site play an important role in the implementation of **move-join**. These numbers are obtained from the position of each key in the agent site. For the source keys, their minor group number is their position in the agent segment. For destination keys, it is their position minus the number of source keys preceding

```
move-join(M, source.data, dest.data) {
    asd.jlen = asd.scnt * asd.dcnt;                  //length of join segments
    joinlength = plus-scan(asd.jstart, asd.jlen);    //total length of join vector
    join = new(joinlength);                          //new vector set length

    seg-index(agent.segindex, asd.len);              //enumerate segment sites
    //Distribute info from asd to agent, so that it can
    //be forwarded to the source and destination
    seg-distr(agent.<dcnt,scnt,jstart>, asd.len, asd.<dcnt,scnt,jstart>);
    //Send info home to SOURCE
    cond-send(source.<minor,jstart,dcnt,scnt>, agent.home,
            agent.<segindex,jstart,dcnt,scnt>, ! agent.dest);
    //Send info home to DEST
    cond-send(dest.<minor,jstart,dcnt,scnt>, agent.home,
            agent.<segindex-scnt,jstart,dcnt,scnt>, agent.dest);

    //Calculate start positions in join vector: notice complementary computation
    source.jsite = source.jstart + (source.dcnt*source.minor);
    dest.jsite = dest.jstart + (dest.scnt*dest.minor);

    //Send start-bit to seg descriptors
    join.ssd = 0;
    cond-send(join.ssd, 1, source.jsite, source.active);
    plus-scan(dest.rstart, dest.scnt);               //result-start
    join.dsd = 0;
    cond-send(join.<rstart,dsd>, dest.<rstart,1>, dest.jsize, dest.active);
    seg-copy-scan(join.rstart, join.rstart, join.dsd); //copy across
    seg-index(join.segindex, join.dsd);              //segmented index wrt dsd
    join.rpos = join.rstart + join.segindex;         //add segindex to it

    //Transpose the destination addresses within segments
    seg-transpose(join.rpos, asd.<jlen,jstart,scnt,dcnt>);

    //Save fields necessary for moving data in the mapping
    ADD-TO-MAPPING(M, source.jsite, dest.<jsite,scnt>, join.<ssd,rpos>);

    //Data movement is now simple
    cond-send(join.sdata, source.data, source.jsite, source.active);
    copy-scan(join.sdata, join.sdata, join.ssd);
    //Permute source data to final position
    send(join.sdata, join.sdata, join.rpos);

    //Distribute data for dest
    seg-distr(join.ddata, dest.data, dest.scnt);
}
```

Algorithm 8.5: The implementation of the **move-join** operation in the EREW scan vector model. All operations are unit step, but the length of the join vector is related to the product of the source and destination vectors.

```
//Each segment is treated as a row-major ordered array of size rowsize × colsize.
seg-transpose(join.data, asd.<jlen,jstart,rowsize,colsize>){

    //distribute data over segments
    seg-distr(join.<rowsize,colsize,start>, asd.<rowsize,colsize,start>, asd.jlen);

    //Enumerate sites of segments
    seg-index(join.segindex, asd.jlen);

    //Calculate new segment-wise address
    join.row = join.segindex / join.rowsize;
    join.col = join.segindex % join.rowsize;
    join.newindex = join.row * join.colsize + join.col;

    //Add segment-start to new index
    join.newindex += join.start;

    //Send to new position
    send(join.data, join.data, join.newindex);
}
```

Algorithm 8.6: The segmented-transpose function. Some simple address arithmetic suffices to compute the permutation vector that represents the transposition of each segment.

them in their group, scnt.

Fields asd.scnt and asd.dcnt store information pertaining to entire groups. The join segments for each group are described by asd.jlen and asd.jstart. This information is moved back to each of the source and destination key sites by first distributing it over the agent and then sending it home. The field source.dcnt then gives the number of destination sites that each source matches, and hence the number of times it must replicate its value. Each source key also has a copy of the start of its join segment in source.jstart.

Each source must calculate the index of the site in its join segment that it sends its value to in field source.jsite. The computation of source.jsite is performed elementwise in a simple arithmetic statement. Minor group numbers are necessary in this computation so that each source key sends their value to a different site in their join segment. The destination-count of each source, source.dcnt, is the spacing factor that is used for computing the scattering pattern. The sites indicated by source.jsite also become the beginning sites of the segments for source.sdata. A start-bits segment descriptor is created by sending a "1" value to those sites referenced by source.jsite. This relationship was pointed out in Figure 8.8.

A similar computation is performed in the destination to compute dest.jsite and the segment descriptor join.dsd. The two start-bit segment descriptors join.ssd and join.dsd describe the partitioning of the join segments in the two complementary ways that are related by the transpose operation.

8.7.3 Complexity

All of the operations required to implement **move-join** are unit step operations in the vector model. Hence, its step complexity is $S = O(1)$. However, the work complexity of this operation is dictated by the sum of the sizes of the join segments, which are the product of the number of source and destination sites in each group. This leads to a $W = O(s \times d)$ work complexity measure for **move-join**.

8.7.4 move-join with the special mappings

The mappings created for the special cases of a self-match or **collapse** described earlier can be used with **move-join**. For the self-match, the algorithm remains largely unchanged, but wherever fields associated with the destination vector set are referenced, the corresponding field from the source is substituted. This simplifies its implementation a slight amount.

When **move-join** is applied to a mapping produced by **collapse**, a curious thing occurs. Because a collapse mapping gathers up source sites to a single destination, and **move-join** expands each destination out for the sources mapped there, the result is a simple permutation of the active source sites. No extra copies of any of the source values are required. This follows from the fact that the dcnt value of each source site is 1 or 0 (if it was a **null** key). The procedure for calculating the permutation vector for the source values is relatively straightforward.

8.7.5 Implementing the collect operator of Paralation Lisp

Paralation Lisp supports segmented vectors as first-class data objects. Its **collect** function used a mapping and source field to produce a nested segmented vector of the destination. In our notation we must represent a segmented vector as a pair of fields: a data field, and a segment descriptor.

The comparison of **collect** to our **move-join** is simple. The result data field produced by **collect** is our field result.s, and its segment descriptor is dest.scnt. Our **move-join** procedure computes these fields but presents them in a different manner.

Chapter 9

Implementing match using hashing

This chapter describes an approach to the implementation of **match** and **move** using parallel hashing. Because hashing requires concurrent-read and combining-write operations this method of implementation is classified as requiring variations of the CRCW vector model. While simulation of CRCW primitives is possible on the EREW vector model, such simulations are expensive and increase the bounds presented here. Whereas the approach of the previous chapter used segmented-scan instructions to combine values belonging to the same group, this approach makes use of the combining-send operations that are supported in the CRCW scan vector model. Instead of sorting keys to bring them near one another in a segment, parallel hashing is used to allow all of the members of a group to find a common site with which to use the combining-send operation.

Parallel hashing is well-known in the folklore of parallel computing, but there has been a remarkable dearth of literature describing it in detail. This chapter begins by first describing parallel hashing as the solution to a more general problem called the "naming problem," and then discusses the performance the algorithm obtains on the Connection Machine CM-2 and the CRAY Y-MP. In the sections following, new algorithms for implementing **match** and **move** are described using the parallel hashing algorithm.

9.1 Parallel Hashing

Hashing techniques have long been used to perform table lookup on serial machines. While some hashing techniques have also been extended to parallel and vector computers [Kan90], it is only recently that the algorithmic complexity of parallel hashing algorithms has been studied[LPP91, And88]. In [Kan90], a vectorized algorithm was described that is similar to the one presented here, however no bounds were placed on the expected performance of the algorithm. While [LPP91] provided the framework for understanding the total number of parallel steps required, and [And88] provided an empirical study of parallel step counts, neither recognized the importance of work complexity. In this section we place expected case bounds on the parallel step count of our algorithm as well as its work complexity. We show that an important indicator of actual performance is the total amount of *work* performed, and that to achieve high performance a parallel algorithm must be carefully designed to perform no more total work than necessary.

We also extend parallel hashing to the solution of a more general problem called the *naming* problem. Algorithms that solve the naming problem may be based on either sorting or hashing. When a total order of the elements to be named is not required, the sorting algorithm performs more work than necessary. This is borne out by the algorithmic complexities of the two methods in which general parallel sorting performs $\Omega(n \log n)$ work for arbitrary sized keys but our hashing algorithm performs only $O(n)$ expected work.

The following sections are organized as follows. First, the naming problem is defined as a generalization of hash table insertion and a number of fundamental parallel algorithms are outlined that use the naming algorithm as a core step. The basic parallel hash table insertion algorithm is then presented along with some variants that attempt to reduce the total number of steps required by the algorithm. The expected case behavior of the parallel algorithm is analyzed and we show that our parallel hashing algorithm performs no more work than its serial counterpart. Finally, we present performance data collected from an implementation of our algorithm on the Connection Machine CM-2 and CRAY Y-MP and discuss the implications of work efficiency with regard to parallel step count.

9.1.1 The Naming Problem

When building a table using hashing, each insertion of a new data item involves finding either a previous instance of the same key, or an empty site in which to put the key. A parallel algorithm for key insertion may be constructed in which a vector of keys are inserted together. The value returned by such a parallel algorithm is a vector of table indices identifying the final hash sites of each key.

This problem attempts to find a mapping of keys from a potentially large universe to a small hash table. This is an instance of the *naming* problem which is as follows: a multiset X of n elements selected from some universe U are to be assigned names in the range $\{1..m\}$, $m \geq n$, such that two elements are given the same name if and only if they are the same element from U.

The naming problem is one that is fundamental to many parallel algorithms. Often the names represent a limited resource and the keys label data that are to share those resources. In the most common instance of the naming operation, the names to be assigned are processor IDs of the available processing sites and keys identify data that are to be combined at a common site. The processing site assigned to common keys is called an *agent*, and does work on behalf of the keys assigned to it. The naming operation serves to separate the construction of a communications mapping from the operation performed with that mapping.

Algorithms

In many applications, there is a total order defined for the keys and the naming operation may be implemented by using a sort procedure. In this method, after the keys are placed in sorted order, they are divided into groups such that a new group begins where a key differs from its left neighbor. The first element of each group is called its "leader." The available names are distributed to the leaders, which then distribute the names to all members of their groups.

The sorting step dominates the time spent in the procedure outlined above, and may not be applicable if there is not a total order on the keys. Even when applicable, because sorting requires $\Omega(n \log n)$ operations it is not work efficient for the naming problem.

This section presents a family of efficient parallel algorithms that solve the naming problem by using parallel hashing. The first step of the algorithm uses a hash function, h, that uniformly distributes the n keys across the name space. At each step of the iterative procedure, a large number of keys claim names. These keys are removed from the working set by using a "packing" step so that the working set size decreases as the algorithm proceeds. Many keys find names in just the first few parallel steps, after which, the number of keys remaining in the working set decreases quickly.

9.1.2 The Parallel Hashing Algorithm

Traditional serial hashing algorithms are concerned with the efficient creation of a hash table (inserting all n elements) as well as being able to locate keys already in the table [Knu68]. The naming problem only concerns itself with the insertion phase of hash table use; its final result is the set of key destinations.

Closed table hash algorithms begin with a function h that is designed to map each of the keys uniformly to some number in the range $\{1..m\}$ for the given m. As each key is inserted, its hash location is probed to examine whether another key has already been inserted there. If so, an iterative search procedure is used to find an empty site. The sequence of hash sites examined by each key is called a "path." Ideally, each key examines a different path so as to avoid a situation where many keys collide repeatedly along the same path. This phenomenon is called "clustering."

In the parallel version of hashing, the insertion of many keys is attempted in one step. Those that fail all examine the next site in their path in parallel. The process continues until all keys find a hash site.

Whereas the analysis of the running time of the serial algorithm is concerned with the *average* time of insertion for each of the n keys, the parallel algorithm must consider two measures of performance. The first is the total iteration count, which is governed by the *maximum* number of probes required to insert any of the keys since the algorithm cannot terminate until all keys find a hash site. The second measure is the total work performed, which is the sum over all iterations of the lengths of the vectors involved in each operation.

Our algorithm accepts a vector field of n keys and returns a field of names in the same vector set. A hash table of length m is allocated and initialized so that all locations have the special value EMPTY. Upon completion, the name for each key is located at the same site in field names.

Pseudo-code for our algorithm is presented in Algorithm 9.1. In the initialization step, the keys are copied to a new vector set called active and the initial hash sites are calculated by **hash-fn**. The home vector records the starting location of each key. After each iteration of the algorithm, these temporary vectors are adjusted to hold only the information relevant to those keys that have yet to find a hash site.

The main step of the algorithm is described by the body of the **while** loop. In parallel, all keys check to see if their hash site is EMPTY. Those that find an empty site try to claim it by sending their key there. When multiple keys are sent to the same site, one overwrites all others ensuring that at least one key claims the site. With one more table reference, the hash site of each key is examined to determine which key claimed it. Keys that find their own value are marked done and their hash site is sent home to the names field. All other keys continue with further probe steps by calculating a new hash site in the last **elwise** statement.

The use of the **pack** step on the active vector ensures that this procedure is efficient by producing a new vector that holds only the elements that are not flagged as done. Procedure **pack** is implemented by first enumerating the **true** sites and then sending the values at these sites to their enumeration address. In the parallel vector model it is implemented with a single **add-scan** operation and a parallel vector permutation. Thus, its complexity is $S = O(1)$ parallel steps and $W = O(n)$ work for a vector of length n.

The function **next-fn** may be designed to implement a number of different probe strategies. The path that each key follows may be defined by either "Linear Probing" or "Double Hashing" probe strategies. Linear probing simply searches the hash sites in sequence. Double hashing attempts to reduce "clustering" by using a probe strategy where the path each key follows is a function of its value. To implement this algorithm, a second hash function, h_2, is employed and the ith site examined for the key is $((h(A) + ih_2(A)) \bmod m)$.

Another variation also adopted in many serial hashing algorithms is the use of an "Overflow" hash table. In this approach, two hash tables (designated T_1 and T_2) are used such that each key makes its first probe into T_1 and makes all other probes in T_2. This arrangement takes advantage

```
hash-insert (key.<key,name>, table.table, m) {
    active.key = key.key;                          //Initialization
    active.home = key.#;
    elwise(active) {
        hash = hash-fn(key, m);                    //compute initial hash site
    }
    while (length(active) ! = 0) {
        get(active.get, table.table, active.hash);     //Examine hash site
        active.enable = (active.get == EMPTY);   //Key is enabled if hash site not claimed
        cond-send(table.table, active.key, active.hash, active.enable); //Attempt to claim it
        cond-get(active.get, table.table, active.hash, active.enable); //examine again

        active.done = (active.key == active.get);  //Key is done if claimed the site
        //Send home info for done keys; pack the others
        cond-send(key.name, active.hash, active.home, ! active.done);
        active.<key.home,hash> = pack-vect(active.<key,home,hash>, ! active.done);
        elwise(active) {
            hash = next-fn(hash, key);             //compute next site
        }
    }
}
```

Algorithm 9.1: Implementing the **match** operation using CRCW scan vector primitives. Most of the work is performed by the parallel hashing procedure, which supplies a unique name for each group of equal keys.

of the fact that many keys require only one probe, leaving table T_2 nearly empty.

In the next section, we show that for Linear Probing, our algorithm performs a total of $S = O(\log n)$ expected steps, and $W = O(n)$ expected work. Analyses of double hashing algorithms show that on the average, they reduce the total number of probe sites examined [Knu68]. We implemented a version of double hashing and compare our results to linear probing on the CM-2. On the CRAY we implemented both probe strategies with and without an overflow table. The performance of our implementations is discussed after an analysis of the algorithmic complexity of the algorithm.

9.1.3 Analysis of the Algorithm

The behavior of the parallel hashing algorithm for linear probing will be analyzed by comparing it to its serial counterpart. The total work, W, of the parallel hashing algorithm is the sum of the lengths of the work vector over all iterations. Since each key remains in the active vector for only as many iterations as it requires probe steps, the total work is the sum of the number of probes required to insert all n keys. When the hash function and hash table size are the same for the serial and parallel algorithms, the total number of probes required by each of the algorithms is shown to be the same.

The number of iterations required to insert all of the keys could conceivably be n for a particularly bad arrangement of keys, however this is almost never the case. An expected bound on the number of iterations required will be made by examining the expected number of probes required to insert the worst-case key.

9.1.4 The total work required to insert N keys

A surprising property of the serial algorithm for Linear Probing was first pointed out by W. W. Peterson [Pet57]:

Theorem 4 *The total number of probes required to insert N keys remains unchanged regardless of the order in which the keys are inserted in the table.*

Proof: (due to Knuth, [Knu68]) The keys are presented in some order $A_1, A_2, ...A_N$. It suffices to show that the total number of probes needed to insert the keys is the same as the total number needed for $A_1...A_{i-1}A_{i+1}A_iA_{i+2}...A_N$, $1 \leq i < N$. There is clearly no difference unless the $(i+1)$st key in the second ordering falls into the position occupied by the ith in the first ordering. But then the ith and the $(i+1)$st merely exchange places, so the total number of probes for the $(i+1)$st is decreased by the same amount the number for the ith is increased.

□

A similar theorem holds for inserting keys in parallel.

Theorem 5 *The total number of hash sites examined remains unchanged if two or more keys are inserted in parallel.*

Proof: Consider two keys presented in order A_1, A_2. If, in parallel, the two keys begin their probes at two different locations, then because they each move only one site forward each iteration, the two keys will find the same empty hash sites as they would had they been inserted in sequence. If the keys happen to begin at the same site, then the first two empty sites past their starting point are the two sites that the keys will find. The two possible arrangements of these two keys into the two sites are exactly the two arrangements dictated by the serial presentation order A_1, A_2 and A_2, A_1. Thus, parallel insertion has the same effect as a random exchange of the order in which keys are inserted by a serial algorithm, which by the previous theorem does not affect the total number of probe steps made.

□

These two theorems state that the parallel linear probing algorithm performs exactly the same number of probes as its serial counterpart. The analysis of the numbers of probes required to insert n elements into a hash table using linear probing is well understood. From [Knu68] we have the following: The average number of probes required to insert the ith key is $C_i' = \sum_{1 \leq r \leq m} rP_r(i)$ where $P_r(i)$ is the probability that the ith key requires r probes to find an unoccupied site. The average number of probes to insert all n keys is $C_n = \frac{1}{n}\sum_{0 \leq i \leq n} C_i'$. The total work is simply $W = nC_n$ for n keys. Knuth gives the value of C_n for linear probing as $C_n \approx \frac{1}{\alpha}\log\frac{1}{1-\alpha}$ where $\alpha = \frac{n}{m}$. Thus, when the hash table is sized so that $m = \Theta(n)$, the expected work performed by the parallel algorithm for linear probing is

$$W = nC_n \approx n\frac{1}{\alpha}\log\frac{1}{1-\alpha} = O(n).$$

9.1.5 The total number of iterations required to insert N keys

The Longest Length Probe Sequence (LLPS) quantity associated with a hashing algorithm measures the longest sequence of probes needed to locate any of the n keys inserted in the table. The LLPS measure is a random variable whose maximum value is obviously n for linear probing. However its average value over all possible arrangements of keys in the hash table, the average LLPS, is also of interest.

Since the parallel linear probing algorithm requires as many iterations as the longest length probe sequence for any key, the average LLPS figure of the serial algorithm measures the average number of iterations that our parallel hashing algorithm requires. Gaston Gonnet [Gon90] has derived bounds on the average LLPS for linear probing and shows that it is $O(\log n)$, when $m > n$. Thus, the parallel step complexity for n keys is

$$S = O(\log n).$$

The same expected bound was given for parallel uniform hashing in [LPP91]. Our experience shows that the average number of iterations required for the hashing algorithm to complete grows slowly with the number of keys. Data presented later shows that the insertion of one million keys is achieved in less than 20 iterations, on average. However, the work figure is a more accurate measure of the total amount of time required by the algorithm.

9.1.6 Implementation on the Connection Machine CM-2

We implemented our algorithms on the Connection Machine model CM-2. Through the virtual processor (VP) mechanism of the Paris assemply language we were able to support the varying sized vectors that our **pack** procedure produced. As stated earlier, this was necessary to ensure work efficiency.

Our choice of a hash function was guided by the target architecture for implementation. One particularly good hash function is based on a result from coding theory [Gal68]. It treats an n-bit key as a polynomial in GF(2^n) and computes an m-bit hash value as the remainder of division by a primitive irreducible polynomial of the field GF(2^m). These hash functions are often not used on conventional architectures because of the bit manipulations that must be performed. Because the CM-2 is a bit-serial machine, these hash functions are extremely fast. Also, because these hash functions require a table whose size is a power of two, they are perfectly suited to the size of VP sets and guarantee that keys are evenly distributed over all processors of the CM-2.

A minor modification was made to the insertion procedure. Because communications operations can require as much as 1000 times as much time as elementwise operations on the CM-2, the procedure **parallel-insert** was modified to reduce the total number of communications operations performed by omitting the first **get** step. Instead, as hash sites are claimed, a flag is set. During the probe step, all keys send their values regardless of the state of their hash site. Then, non-empty hash sites overwrite the value sent there with the key that previously claimed the site. Even with large hash tables, these elementwise operations take virtually no time, and are outweighed by the savings from eliminating the extra communications operations. Theoretically, this increases the work complexity of the algorithm by $O(m \log n)$, but does not have an effect when $\log n < 1000$, which is a reasonable assumption for almost problem size hoped to be solved on a real machine.

Results

A series of trials were taken for n ranging from 50,000 to 2,000,000 elements on a 16k processor CM-2. The key size in this trial was 32 bits but the universe of possible keys was restricted to be a random set of size n. In this manner, the keys represent a random mapping of a set into itself. Each trial point was repeated a total of five times and the average time reported was measured

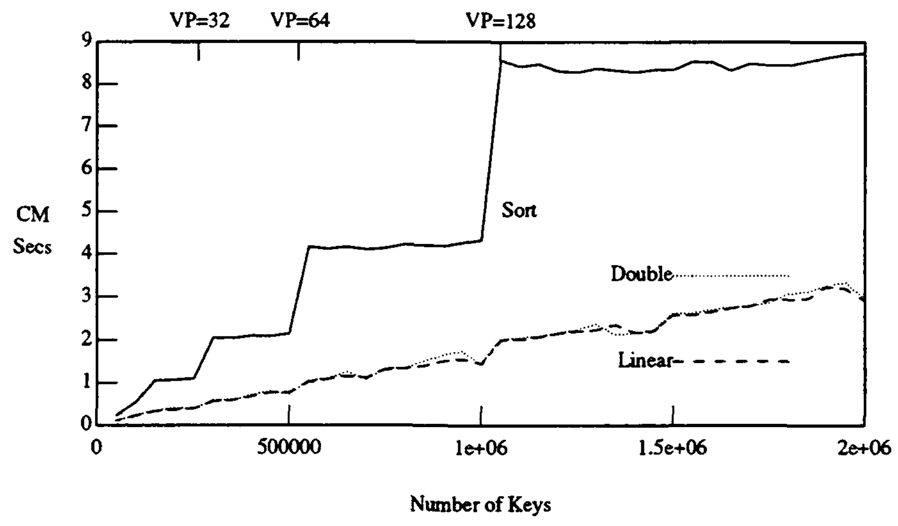


Figure 9.1: A comparison of the hashing algorithms to a naming algorithm that uses sorting. Through all trials the hashing algorithms performed better by a factor of 3 to 4.

in CM seconds, the amount of time the Connection Machine required to complete all iterations. Both "linear probing" and "double hashing" schemes were compared to a naming algorithm that sorted the keys using the RANK function of the system software and enumerated the leaders to assign names. Figure 9.1 summarizes our results.

Through all trials, the hash based algorithms outperformed the sorting based algorithm by a factor of 3 to 4. The sharp jumps in times for the sort based algorithm occurred where the vector size increased past a VP ratio boundary. Our data spans VP ratios of 2 to 128 and shows that the hashing algorithms have linear characteristics even over VP ratio boundaries.

It is interesting to notice that the two variants of hashing require almost the same time for all key set sizes. Even though the total number of iterations required by the "linear probing" variant *always* exceeded those required by "double hashing", the extra iterations were performed on such small vectors that the additional time is negligible. This point deserves further examination.

Figure 9.2 shows the average number of iterations required for each algorithm for different input set sizes. The sharp drops in the curves are due to the fact that the hash table and overflow table sizes were rounded to the next power of two when the key set size exceeded a VP ratio boundary. In those situations when the hash table was very large with respect to the number of keys, the number of iterations became quite small. While we see that double hashing keeps the number of iterations performed smaller than by using the linear probing method, the total times spent by the algorithms are roughly the same, and are not affected very much by the occasional large jumps in the number of iterations.

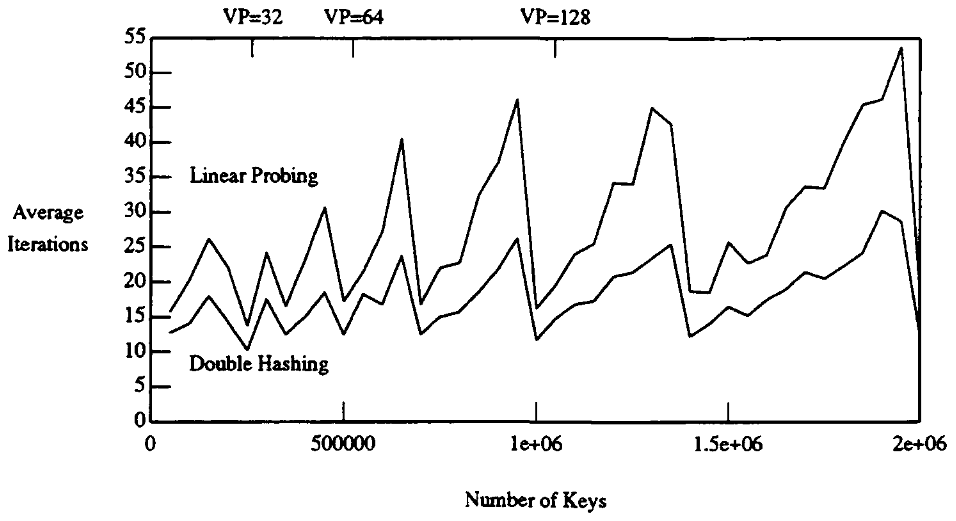


Figure 9.2: The average number of iterations required by the two hashing algorithms. The data show that double hashing keeps the total number of iterations lower than the number required by linear probing. The unusual drops in the curves occur when the hash table increased to the next VP set size on the CM, resulting in a hash table with a very low load factor.

Figure 9.3 provides further insight into the impact of the total number of iterations on the completion time. The trials for the insertion of 1,900,000 keys using double hashing required a wide range of iterations to complete. Even as the iterations increased by almost a factor of 2, the total time grew by only 7%. The time is dominated by the first few probe iterations with long vectors; all other iterations are performed with very short vectors and require a small percentage of the total time.

The decrease in the number of active keys is shown in Figure 9.4. When inserting one million keys using the double hashing algorithm, by the eighth iteration the total number of active keys fell below 100. Probe iterations performed with small vector lengths are completed quickly.

9.1.7 Implementation on the CRAY Y-MP

The implementation of the hashing algorithm on the CRAY Y-MP was straightforward because all operations completely vectorize. We used the CVL (C Vector Library) developed by Guy Blelloch at Carnegie Mellon University for most of the vector operations [BCSZ91], but developed our own hash function in CRAY vectorized C.

As with the implementation for the CM-2, the hash function was highly optimized for the target architecture. We chose to use a standard remainder operator as our hash function and sized the hash tables to be the least prime number greater than the number of keys. However, because

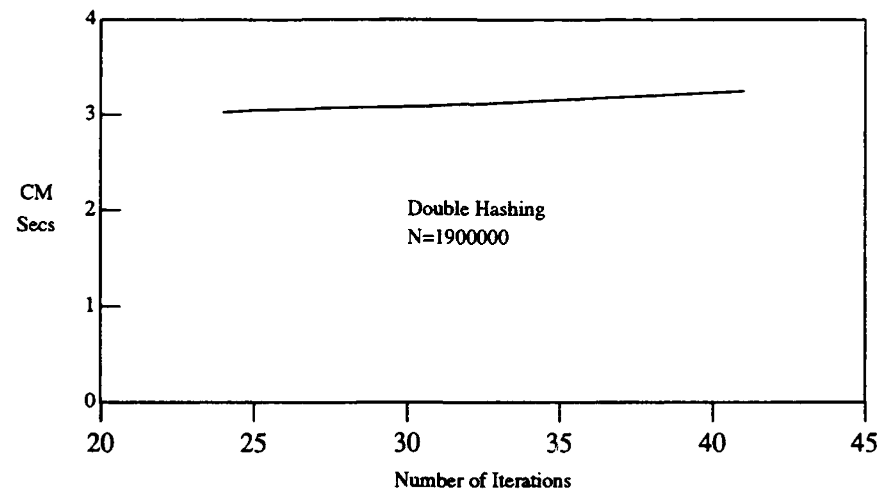


Figure 9.3: A comparison of the execution times when the number of iterations varied greatly. Even though the number of iterations spanned a wide range, the time increased by only a small amount when the number of iterations grew large. All of the extra iterations were performed with very small vectors and required very little time.

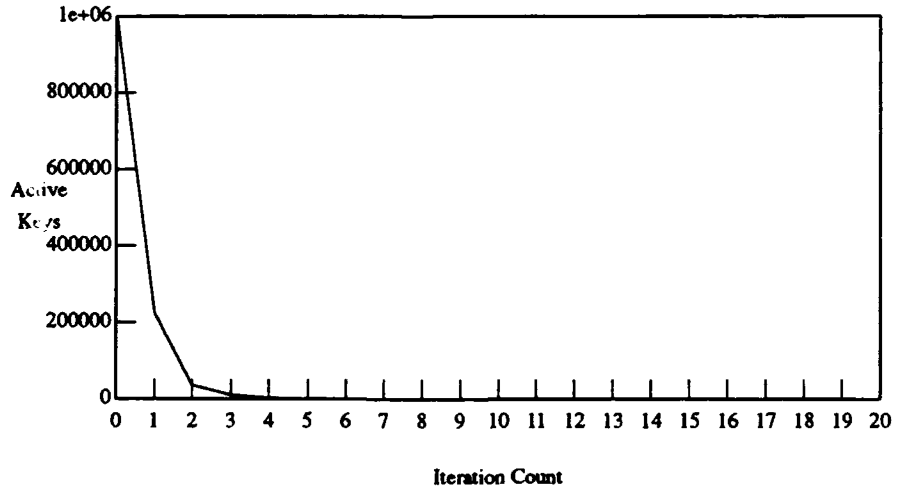


Figure 9.4: The number of active keys at each step in the hashing algorithm. The hashing algorithm eliminates many keys from the active set in just a few iterations. In this graph, the total number of active keys is less than 100 after eight iterations. Most of the iterations are performed with very short vectors.

the standard modulo operator ('%') of CRAY C calls a subroutine that computes the remainder between two 64-bit integers, it was much too slow. We wrote a highly optimized modulo function that took advantage of the facts that the same modulus was applied to all keys, and that it had far fewer than 64-bits of significance. Using these observations, we divided the computation into two parts that each used an integer division and a subtraction operation, yielding a very fast hash function.

The next-site function, **next-fn**, was written to allow the choice of Linear or Double probing, with or without an overflow table. Double probing was described earlier. To implement an overflow table, we conceptually divided the hash table into two parts: it's lower half acting as T_1, and the upper half acting as T_2. The next-site function was written to supply a first probe site in the lower half, with all other probes indexing sites in the upper half. In this manner, a single site address could specify one of the two tables and a site within that table.

Results

Once again, we measured the performance of our hashing algorithm relative to sorting. This time, the sorting time measured represents *only* the time required to sort the keys using the ORDERS subroutine of the system software [Cra88]. The key size in this trial was 64 bits and the keys once again represented the random mapping of a set into itself. Figure 9.5 summarizes the performance for each of the four combinations of probe strategies relative to sorting. Throughout all trials, the hashing algorithms outperformed the CRAY ORDERS subroutine by up to a factor of 5.

It is interesting to notice that the total times required by each of the four approaches to hashing are nearly equal. In general though, the fastest approach was the simple Linear Probing search, while the slowest was Double Hashing with an Overflow table. These results are somewhat surprising in light of the total number of iterations required by each approach.

Figure 9.6 displays the average number of iterations per key required by each of the variations. Clearly, the simple Linear Probing approach performs far more steps than when using Double Hashing with an Overflow Table. The effect observed here is that the extra complexity of the next-site function (**next-fn**) when using Double Hashing and an Overflow table far outweighs any benefits achieved by a reduction in the number of iterations. Thus, the simple and straightforward Linear Probing approach is the fastest even with the cost of additional iterations. While the more sophisticated probing algorithms did reduce the total number of probe steps required, their extra complexity caused a loss in actual performance.

This effect is due to the fact that computation and communication are so nearly balanced on the CRAY Y-MP. While some machines perform local arithmetic operations much faster than they can permute a vector, on the CRAY Y-MP these operations require nearly the same amount of time per element. Because of this, it is faster to perform the simple Linear Probing hashing

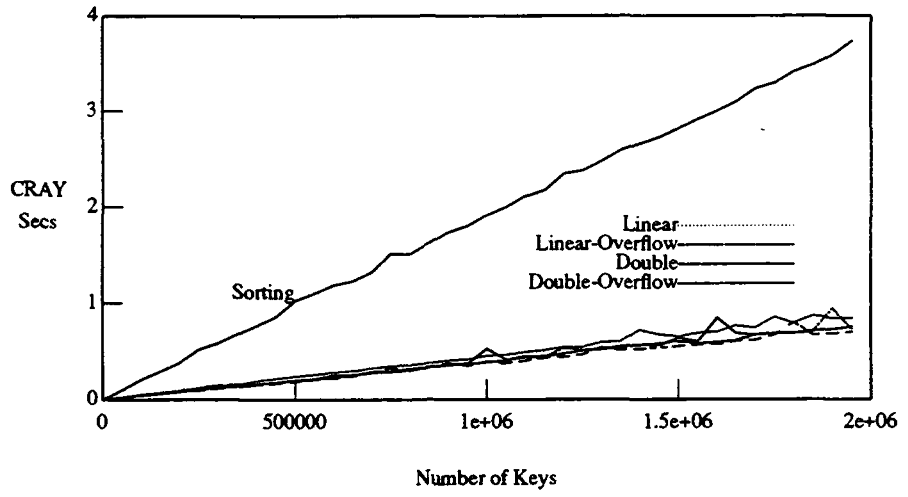


Figure 9.5: A comparison of the times of the hashing algorithms to the time to sort the keys on the CRAY Y-MP. Through all trials, the hashing algorithms performed better by a factor of 4 to 5. While the more complicated probe strategies required fewer total iterations, the total time required by each was nearly equal.

algorithm than to use a more clever (and complicated) probing function. On a machine with slower communication, it might be wise to use the more sophisticated probing schemes.

9.2 Matching by Hashing

The implementation of **match** and the combining **move** operations are especially easy using parallel hashing as the core step. A mapping created using this approach maps all source and destination sites with the same key to the same site in an agent vector. A **move** operation is implemented by all source sites sending their value to the same agent site with a combining **send** where the values of each group are combined. The destination sites then retrieve the combined values using a concurrent-**get**. Figure 9.7 illustrates this type of mapping.

The code that implements **match** is shown in Algorithm 9.2. As before, inactive source and destination sites must be identified as those that do not match other keys. The active fields are used to store this information about each source and destination site. Only non-null keys are initially marked active and they are appended into a work vector along with a flag indicating their type and their home site. A hash table must then be allocated that is bigger than the total number of keys. The parallel hashing procedure is then applied and returns a unique integer name for each distinct key. These are the major group numbers of the keys. With two conditional-send

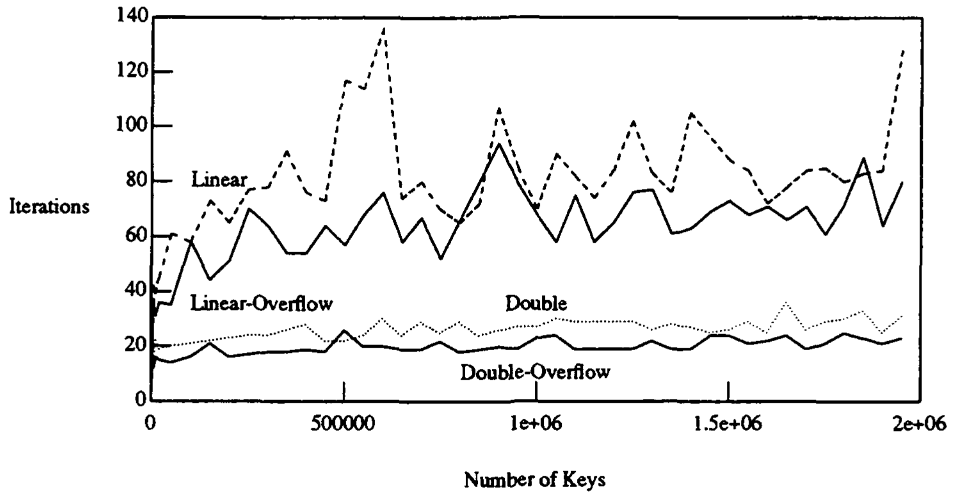


Figure 9.6: The average number of iterations per key on the CRAY Y-MP for each of the four approaches. In general, the more complicated probing schemes reduced the total number of probe steps, as expected. However, as the complexity of the probe function increased, so did the execution time, making the simplest probe scheme preferable.

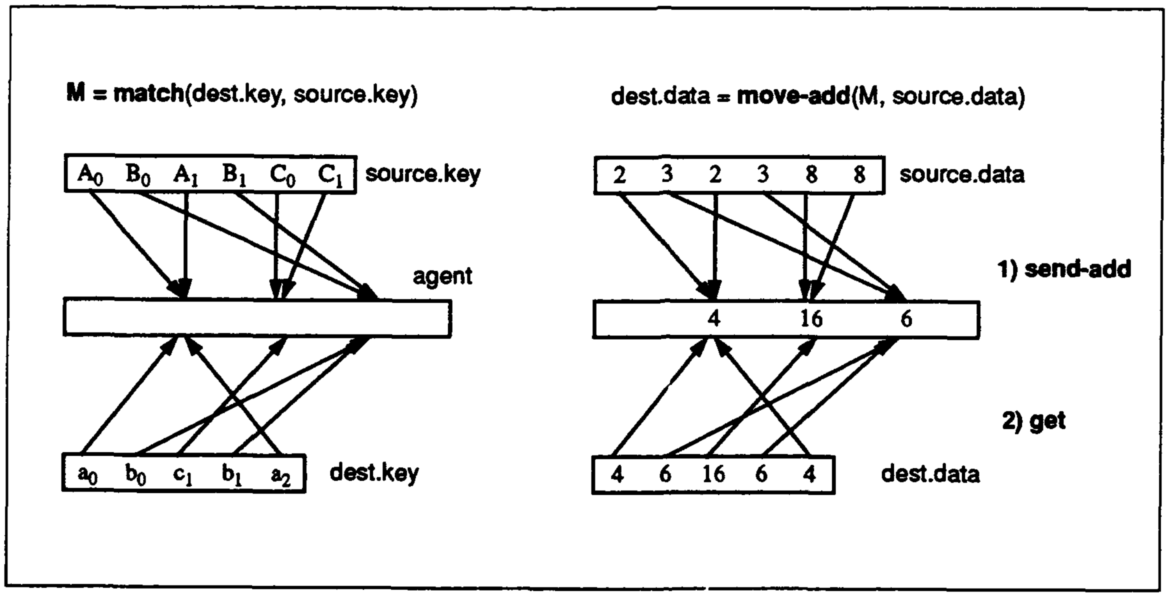

M = match(dest.key, source.key)
A_0 B_0 A_1 B_1 C_0 C_1
source.key
agent
a_0 b_0 c_1 b_1 a_2
dest.key
dest.data = move-add(M, source.data)
2 3 2 3 8 8
source.data
1) send-add
4 16 6
2) get
4 6 16 6 4
dest.data

Figure 9.7: A mapping may be created by using parallel hashing. The mapping data structure directs all equal keys to the same agent site. A **move** operation is executed by combining source values in the agent, and then retrieving copies for each destination.

statements, the names are sent to the homes of each of the source and destination in the name fields. The end result of this procedure is that equal keys in the source and destination belonging to the same group are given the same name. This name is used as an integer index into the agent vector.

Next, using a **send-add**, each agent site obtains a count of the number of source and destination keys in its group in fields scnt and dcnt. These counts are then copied back to the source and destination to detect further inactive sites. A source site is inactive if the number of destination sites in its group is 0, and a destination is inactive if the number of sources in its group is 0. The mapping data structure saves the agent-site names given to the keys, the active information, and the count fields.

Code for the simple combining **move** is shown in Algorithm 9.3. Two main steps implement a combining **move**. In the first step, all active source sites send their values to a field in the agent with the appropriate combining send operation. Then, all active destination sites retrieve the combined values from the agent sites. In the final step, inactive destination sites are set to **null**.

9.2.1 Complexity

The total number of sites involved, $n = s + d$, is the sum of the lengths of the source and destination vector sets. A combining **move** is a unit step operation in the CRCW scan vector model with $W = O(n)$ work complexity because each of its three steps are primitive operations in this model. The complexity of the **match** operation is dominated by the parallel hashing routine. As shown earlier, for n keys parallel hashing performs $S = O(\lg n)$ steps and $W = O(n)$ expected work. However, some important special cases can be exploited that improves these bounds.

Enumerable Fields

When the key fields are marked with the Enumerable property, as described in Chapter 4, there is a unit time function that translates the keys into small integers. These numbers become the names of the keys, and the iterative procedure of the hashing routines may be bypassed. A hash-based mapping on an Enumerable key field can be created in $S = O(1)$ steps. Special cases involving Enumerable keys are those involving a **match** on index fields, Boolean fields, or any small integer field.

Other Integer Fields

For any **match** involving Integer fields, even if they are not marked Enumerable, it is a simple matter to use two reduction operations to find the maximum and minimum keys of the fields. If

```
match (dest.key, source.key) {
    source.active = (source.key ! = null);
    dest.active = (dest.key ! = null);

    work.<key,dest,home> =
        cond-append (source.<key,0,#>, dest.<key,1,#>, source.active, dest.active);

    //Allocate hash table and initialize
    agent = new(Θ(length(work)));
    agent.table = EMPTY;

    //Use hashing to obtain integer names
    hash-insert(work.<key,name>, agent.table);

    //Send "names" back home
    cond-send(source.name, work.name, work.home, ! work.dest);
    cond-send(dest.name, work.name, work.home, work.dest);

    //Count source and dest group sizes
    send-add(agent.scnt, source.name, 1);
    send-add(agent.dcnt, dest.name, 1);

    //Retrieve counts in source and dest
    cond-get(source.dcnt, agent.dcnt, source.name, source.active);
    cond-get(dest.scnt, agent.scnt, dest.name, dest.active);

    //Make additional sites inactive
    source.active &= (source.dcnt ! = 0);
    dest.active &= (dest.scnt ! = 0) ;

    //Save necessary fields in the mapping
    M = MAPPING(source.<name,active,dcnt>, dest.<name,active,scnt>,
            agent.<scnt,dcnt>);
}
```

Algorithm 9.2: Implementing the **match** operation using CRCW scan vector primitives. Most of the work is performed by the parallel hashing procedure, which supplies a unique name for each group of equal keys.

```
move-op (M, source.data) {
    //Send source values to agent using combining function
    cond-send-op(agent.data, source.data, source.name, source.active);

    //Copy value to destinations
    cond-get(dest.data, dest.name, dest.active);

    //Set inactive destination sites to null
    dest.data = dest.active ? dest.data : null;
}
```

Algorithm 9.3: Implementing a general combining **move** using CRCW-PLUS scan vector primitives. A single combining-**send** combines the source values, and a single **get** replicates them for every destination.

the range defined by the two, R = max − min, is small enough so that an agent vector of length R may be allocated, then the translation function simply subtracts min from each of the keys. These values become the names of the keys and the mapping can be created in $S = O(1)$ steps.

9.2.2 Optimizations

Some special instances of **match** should be recognized to enhance performance. While the optimizations discussed here do not affect the asymptotic complexity of the algorithms that implement **match** and **move**, they can be used to significantly reduce the constant factors describing the cost of their implementation.

"NoNulls" fields

When the source and destination fields of a **match** operation both have the NoNulls property, the implementation may choose to avoid tests for **null** key values. The **conditional-append** may be foregone, and a regular vector **append** can be used. On some machines, (such as the CRAY Y-MP) the unconditional append is much faster as it avoids address arithmetic, and becomes two simple block copy instructions. Algorithms on sparse matrices in particular involve no **null** valued keys.

Self-match

As with the sort-based **match** implementation, when the source and destination fields are the same, the **append** step can be omitted so that only half of the keys participate in the hashing procedure. Since hashing dominates the time of the **match**, this optimization can save almost a factor of two over the naive implementation.

Library Function collapse

The **collapse** function can easily be implemented directly by modifying the basic hash-based **match** algorithm. Proceeding as with a self-match, the keys of the source are given names using the hashing algorithm. This step associates each group with a unique site in the agent vector, as shown previously in Figure 9.7. However, because the agent sites occupied may not be contiguous, the name space must be compressed.

Compression is straightforward. In a single parallel step, the occupied agent sites are labeled with a "1" value, and an **add-scan** numbers them sequentially. Then, the source vector replaces its names with the result calculated by the **add-scan**. These new names direct each group to a site in a new vector that has no "holes." As mentioned before, **collapse** has the same complexity measures as a full **match**, but this procedure can save a significant amount of time.

Compression of the name space in this fashion can be applied to the basic implementation of the hashing procedure so that the names produced are contiguous sites in the agent vector. This approach requires only a few extra steps, but can result in a significant saving of space by reducing the size of the agent vector.

9.2.3 Multi-field match and move

Both **match** and **move** are defined so that tuples as well as simple atomic types can be used as arguments. As with the sort-based approach, the simplest way to handle tuple keys is to pack them into a single machine word and perform the **send** and **get** instructions of the **match** and **move** operations. When the fields of the tuples will not fit into one machine word, then all **send** and **get** operations must be extended to multi-word counterparts. Of the combining functions, **arb**, **max** and **min** have multi-word counter parts. As discussed earlier, **move-add** does not generalize to carrying overflow information over to the next fields. Because this would require calculating the overflow due to an addition beyond what can be represented in a single machine word, such a capability is beyond what most machines can provide.

For multiple words, the extension of the **get** operation is trivial. A **get** on a w-word tuple becomes w separate **get** operations, adding $S = O(w)$ steps to the algorithm. The **send** operations of the parallel hashing, **match** and **move** algorithms must be extended to their multi-word counterparts. This can also be done adding only $S = O(w)$ steps to the algorithms. Because w will be a constant for a given algorithm, these changes do not affect the asymptotic analysis of the algorithms given earlier, but only affect their absolute performance.

Code for a multi-word **send** appears in Algorithm 9.4. It is generalized for any of the combining **send** operations **send-arb**, **send-max** or **send-min** where the letters "op" are replaced by the appropriate collision handling strategy. Beginning at the most significant word, all sites send their value to the destination using the combining method indicated. Then, all sites retrieve

```
multi-word-send-op (w, dest.d[w], source.d[w], source.a) {
    for (i = (w-1) to 0) {
        source.enabled = 1;
        //send a word if still enabled
        cond-send-op(dest.d[i], source.d[i], source.a, source.enabled);

        //examine arbitrary value written
        cond-get(source.get, dest.d[i], source.a, source.enabled);
        source.enabled &= (source.get == source.d[i]);
    }
}
```

Algorithm 9.4: Simulating a multi-word **send** using **send** and **get** operations. After sending each word, each site examines the value that was written. If the value written is the same as the one at that site, then the site is still enabled for the next word.

the value written. If the site matches the value written there, then it remains enabled for the next word. Note that this function can be turned into a **cond-send** by initializing source.enabled to only those sites that should be active.

9.3 move-join

The implementation of **move-join** is somewhat simpler using CRCW primitives than it is for the EREW model, but has many concepts in common with it. The use of minor-group numbers is much the same, but it is much easier to calculate them directly using a **multiprefix** instruction than it is to get them from segment-indices in the agent vector. To replicate the source values, instead of using a segmented **copy-scan** a concurrent **get** operation is used to retrieve the appropriate copies.

The strategy used is to first move the source values of each group into a segment in a special vector set called Scopies (source-copies). Then, the destination sites of each group can find the beginning of the segment of copies for its group and can calculate index offsets to find the other copies of its source values. Using a concurrent **get** operation, as many copies as necessary are moved to the final sites.

Algorithm 9.5 describes the implementation of the **move-join** operation. The fields created during its execution are illustrated in Figure 9.8. For each group, there is one site in the agent vector. In the first steps, each agent site calculates the start position of the copies of the source values in a new vector set by performing an **add-scan** on the agent.scnt values. The result is the vector agent.Sstart (source-start-position). Each of the source and destination vectors retrieve these values so that all sites know where the copies of the source values for their group begin.

```
move-join(M, source.data, dest.data) {
    scopylen = add-scan(agent.Sstart, agent.scnt);
    Scopies = new(scopylen);                    //allocate temp vector set

    //Source and Dest copy source-start value
    cond-get(source.Sstart, agent.Sstart, source.name, source.active);
    cond-get(dest.Sstart, agent.Sstart, dest.name, dest.active);

    //Source now needs minor group numbers
    source.count = (source.active ? 1 : 0);     //inactive sources count 0
    multiprefix-add(source.minor, agent.scnt, source.count, agent.name);
    source.spos = sourse.Sstart + source.minor; //rpos is a unique position in Scopies

    //For destination, fan-out segment descriptor is dest.scnt
    seg-index(join.segindex, dest.scnt);
    seg-distr(join.Sstart, dest.Sstart, dest.scnt);  //copy Sstart of each group across segs
    join.spos = join.Sstart + join.segindex;

    //Save fields for re-use
    ADD-TO-MAPPING(M, source.spos, dest.rpos, Scopies.#);

    //Move actual source and dest data
    cond-send(Scopies.sdata, source.spos,
            source.data, source.active);        //send source to Scopies
    get(join.s, Scopies, sdata, join.spos);     //get copies for join result
    seg-distr(join.d, dest.data, dest.scnt);    //distribute dest data across segs
}
```

Algorithm 9.5: The implementation of **move-join** using CRCW scan vector primitives. In this approach, the source data is rearranged so that data from the same groups is adjacent. From there, a concurrent **get** from the join sites makes as many copies as necessary. Destination data is copied using a segmented-distribute.

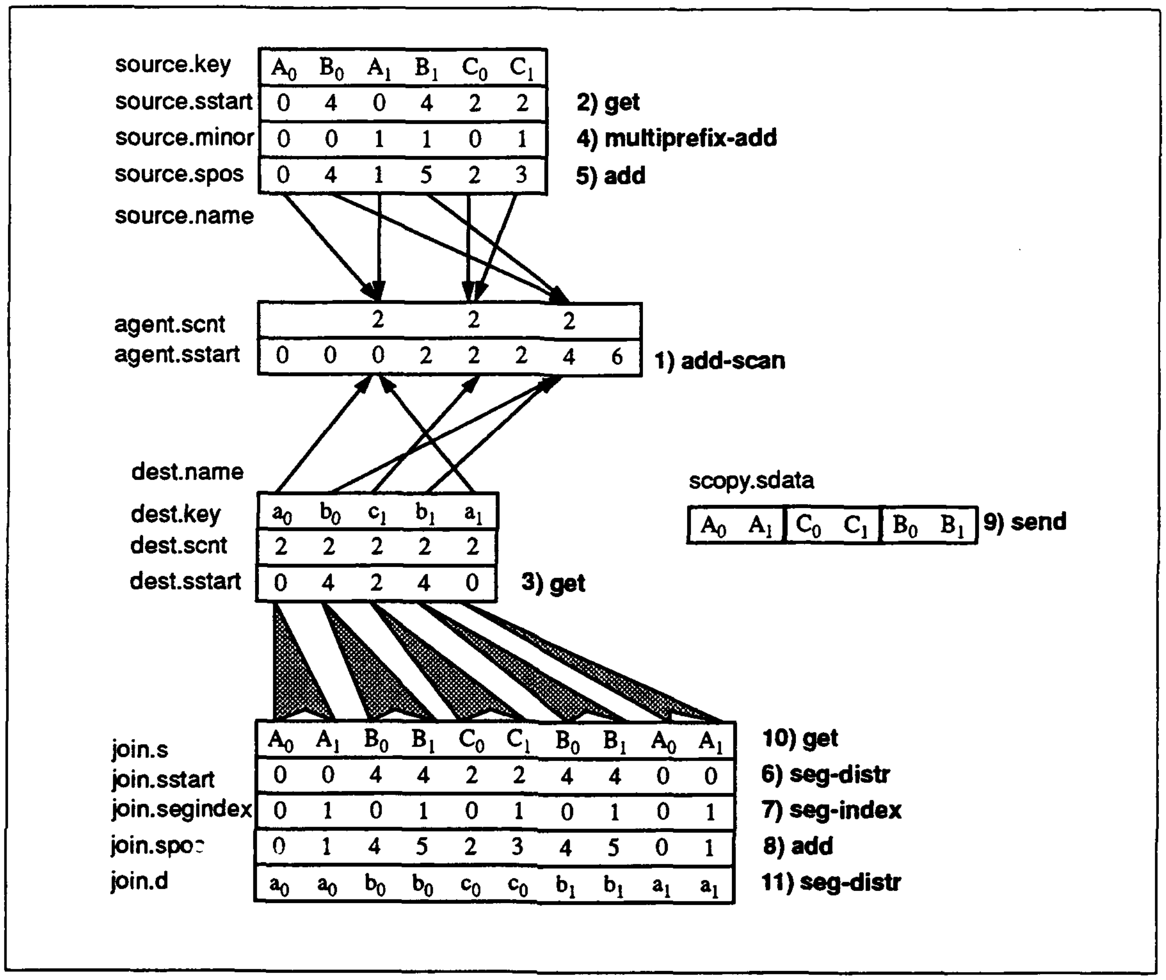


Figure 9.8: The steps of the **move-join** procedure for a hash-based mapping. Copies of the necessary values are made using a concurrent **get** instruction.

Now, each of the source sites has a copy of the beginning of the segment to which they should send their values, but the source members of each group must be assigned unique offsets. This is where the minor group numbers are required: they become the unique offsets for each source member of a group. A single **multiprefix-add** operation suffices to enumerate the source values of each group beginning at 0, providing the minor group numbers. By adding these values to source.Sstart, each source site calculates a unique site in Scopies to send its value.

As before, the join result fans-out from the destination vector, with a segment descriptor described by dest.scnt. This field was calculated previously by **match**. The sites of these segments are enumerated, and then the beginning position of each group in Scopies is distributed across. By adding join.Sstart and join.segindex, each join site calculates the index of the appropriate copy of a source value from its group that it should **get**.

At this point, the address computations are completed, but no actual data has been moved. The necessary fields for data movement are added back into the mapping structure so that further applications of **move-join** with the same mapping can skip the preceding steps.

The data movement of the **move-join** operation proceeds by each source sending its value to the site referenced by source.spos. Then, each join site retrieves the necessary copy using join.spos. The destination values are simply distributed across the segments as before.

9.3.1 Complexity

As before, the **move-join** operation is implemented in $S = O(1)$ steps. Its work complexity is governed by the size of the join vector, which could contain a site for each cross-product. Thus, its work complexity is $W = O(s \times d)$.

9.3.2 Hybrid Approaches

The two mapping types, sort-based and hash-based, provide information about groups in different ways, but they also have much in common. The two different **move-join** algorithms, developed for these two mapping types, may actually be used with either mapping type with appropriate modifications. What differentiates these two algorithms is how they achieve the duplication of source values for each of the join sites. The method of Algorithm 8.5, called **move-join-sort** here uses a **copy-scan** operation to make copies of the values. The second, Algorithm 9.5 which will be called **move-join-hash** here, uses a concurrent-**get** from the join sites to make copies of the values. However either mapping type may be used with a variant of either **move-join** algorithm to construct hybrid approaches. The only substantial change to either of the algorithms is where the minor group numbers are obtained from.

When using a sort-based mapping with the strategy of **move-join-hash**, the minor group numbers must come from the segment indices of the keys in the agent. Alternatively, when using

Table 9.1: Strength of CRCW Model required to support the hash-based **match** and **move** operations. The weakest machine can only support a **move-arb**. The **multiprefix** instruction is necessary for the efficient implementation of the **move-join** operation using a hash-based mapping.

Machine Model	Operations Supported, Cumulative
CRCW-ARB	**match, move-arb**
CRCW-PLUS	**move-add, move-max, move-min**
CRCW-MULTIPREFIX	**move-join**

a hash-based mapping with the **move-join-sort** algorithm, the minor group numbers are computed using a **multiprefix** instruction. Most of the steps of the remainder of the algorithms are largely unchanged.

If a source site matches many destination sites, then many copies of its value are required to implement the **move-join** operation. When using the concurrent-**get** of algorithm **move-join-hash**, these sites would represent hot-spots that could lead to memory contention. On some architectures, such a situation can seriously degrade performance. The use of a **copy-scan** operation to replicate the values in **move-join-sort**, could be much faster. Choosing between the two is highly data dependent, but could be influenced by program analysis or **pragma** statements provided by the programmer.

Notice that both of the hybrid schemes require a CRCW model of computation. Even though a sort-based mapping can be created using only EREW primitives, the **move-join-hash** algorithm requires a concurrent-**get**. Similarly, a hash-based mapping is created using the CRCW model, and either approach to implementing **move-join** requires the **multiprefix** instruction.

9.4 Strength of the CRCW Models

When the CRCW models were introduced, they were categorized as CRCW-ARB, CRCW-PLUS and CRCW-MULTIPREFIX variants. The weakest model is the CRCW-ARB, while the strongest is the CRCW-MULTIPREFIX. With all variants, a hash-based **match** operation can be implemented. However, without some of the more powerful operations, there are limitations to the types of **move** operations that are efficiently implemented using this type of mapping. Table 9.1 summarizes these relationships.

With a sufficiently powerful machine that supports the multiprefix instruction, all variants of the **move** operations are efficiently supported. However, on a less powerful machine, the types of **move** operations possible are somewhat restricted. For instance, algorithms that require a **move-join** can only be implemented using the hash-based **match** if the machine supports a **multiprefix** instruction. Otherwise, the sort-based **match** must be used.

The next chapter discusses the the **multiprefix** instruction in more detail. It presents an algorithm for the **multiprefix** instruction in the general PRAM model and demonstrates that it can be efficiently simulated on a vector processor. For a special case, it shows how the CRCW-ARB PRAM model can simulate the CRCW-MULTIPREFIX PRAM model with no loss of efficiency.

Chapter 10

Implementing multiprefix on Parallel and Vector Computers

This chapter presents a work efficient algorithm for the multiprefix operation on n elements that runs in $S = O(\sqrt{n})$ parallel steps on a $p = \sqrt{n}$ processor CRCW-ARB PRAM. Instead of the parallel vector models, this chapter uses traditional PRAM models in which scan operations are not supported as primitives. In contrast to the vector models, these models make explicit the number of processors with a parameter, p. As with the vector models, these models also assume that a parallel memory access is a unit time operation, and variations of these models provide different memory access collision policies. The CRCW-ARB model ensures only that of multiple processors writing to the same location, an arbitrary one succeeds. We make use of this feature to resolve data dependencies in the first phase of the algorithm only so that all later steps guarantee EREW memory access.

A synchronous PRAM algorithm may be simulated by a vector computer with scatter/gather capability by issuing one vector operation for each parallel step. Using this approach, a fully vectorized version of our algorithm has been designed for the CRAY Y-MP and provides good performance for a number of important algorithms, independent of **match** and **move**. For the integer sorting test of the NAS benchmarks, our multiprefix operation was used to create an algorithm that is competitive in performance with the current best algorithms for that machine. Our algorithm also makes possible the simulation of a CRCW-PLUS PRAM on a p processor CRCW-ARB PRAM with only constant slowdown for problem sizes $n = \Omega(p^2)$. With the multiprefix operation, it is possible to implement a full suite of **match** and **move** operations using a hash-based mapping.

```
A      = [ 1  1  5  1  6  1  7 ]
L      = [ 3  3  2  3  2  3  2 ]

multiprefix-add (S, R, A, L);

INDEX =  1  2  3  4  5  6  7
S      = [ 0  1  0  2  5  3  11 ]
R      = [ 0  18 4 ]
```

Figure 10.1: An example multiprefix result given a vector of data values (A) and an associated vector of labels (L). The two results computed are the multiprefix sums (S) and the reductions for each label (R). Each reduction value represents the sum of all values with that label.

10.1 Background

Each of an ordered set of data values $A = \langle a_1, a_2, ..., a_n \rangle$ are associated with integer labels $L = \langle l_1, l_2, ..., l_n \rangle$ with $l_i \in \{1, 2, ..., m\}$. The multiprefix problem is to compute a collection of partial sums $S = \langle s_1, s_2, ..., s_n \rangle$ for the values and a set of reductions $R = \{r_1, r_2, .., .r_m\}$ for the labels such that

$$s_i = \sum \{a_j \mid l_j = l_i \text{ and } j < i\}$$

and

$$r_k = \sum \{a_j \mid l_j = k \text{ and } k \in L\}.$$

That is, for each value its multiprefix result is the sum of all elements with its same label preceding it in vector order. While for each label, its reduction is the sum of all values with that label. For example, given a vector of values, A, and labels, L, as shown in Figure 10.1, a call to the multiprefix operator produces the results in S and R as illustrated. Because only preceding values contribute to each sum, the first sum of each "group" of equal labels is 0, and because only the labels 2 and 3 appear in the L vector, the reduction vector R has non-zero values only at these positions.

The general multiprefix operator solves the multiprefix problem and extends the summing operation to any binary associative operator on values of arbitrary type. As an operator, it accepts vectors containing the values and labels, and a binary associative operator, producing the values of the sums and reductions with respect to the operator given. Typical operators are **max**, **min**, **add**, **and** and **or** on data types Integer, Double and Boolean. Throughout this chapter, the focus is the **multiprefix-add** operator, but the discussion generalizes to any binary associative operator as long as 0 is replaced with the appropriate identity element for the operator chosen.

The multiprefix operator has been previously proposed as a parallel primitive for the Fluent abstract machine [RBJ88] and as a general purpose parallel primitive [Coh90]. The definition of

multiprefix given here is nearly identical to the one given in [Coh90] but differs from the one given in [RBJ88] slightly. In that formulation, the labels were references to shared variables to which the reduction values were written, and the operation was not presented in a data parallel framework. These differences are inconsequential.

The multiprefix operator subsumes the functionality of many other parallel primitives. For instance, it provides the functionality of the **fetch-and-op** primitive of the NYU Ultracomputer [GLR81]. While the **fetch-and-op** primitive is non-deterministic in its evaluation order, the multiprefix operator ensures that results are computed in vector index order. Multiprefix also provides the functionality of the segmented-scans [Ble90] and the combining-**send** of the Connection Machine [Hil85], and can be used to implement the β operation of CM-Lisp [SH86]. A segmented-scan is simulated by distributing the same label to each element in a segment and then executing the multiprefix operation. A combining-**send** operation is provided directly by multiprefix, but only the reduction values are used. When the multiprefix sums are not computed, this is also called a **multireduce** operation. The multireduce operation occurs most frequently as histogram computation which is important enough that a special "Vector Update Loop" compiler directive has been suggested to identify this procedure [PMM92, page 18]. In short, the multiprefix operation attempts to unify the functionality of these many varied parallel primitives.

10.1.1 Applications of Multiprefix

Multiprefix has been proposed as a parallel primitive because of its generality and the power that it provides for expressing many parallel algorithms. For example, [RBJ88] show how to implement an integer sorting routine using multiprefix in just a few steps. It can also be used as a general histogramming operation when the prefix-sums are not required. Of course, it is essential to the implementation of **move-join** when using a hash-based **match** algorithm.

10.1.2 Theoretical Results

This chapter presents an algorithm for implementing the multiprefix operator for n values in $S = O(\sqrt{n})$ parallel steps on a $p = \sqrt{n}$ processor CRCW-ARB PRAM in $s = O(n+m)$ space. Another important complexity measure of an algorithm is the total work performed and is the sum over all steps of the number of primitive, scalar instructions issued. For a serial algorithm on one processor this is its traditional time complexity. A parallel algorithm is *work efficient* if it performs no more work than an equivalent serial algorithm. Our multiprefix algorithm performs $W = O(n)$ work; hence it is work efficient.

Of the CRCW PRAM models, the CRCW-ARB model assumes only that of multiple processors writing to the same location, an arbitrary one succeeds. While not as restrictive as the EREW model, the CRCW-ARB PRAM model is a fairly realistic representation of many current parallel architectures that provide a shared memory. The CRCW-PLUS PRAM model allows a combining function to be applied to values concurrently written to the same location [CLR89, page 690]. Our multiprefix algorithm can be used to simulate a concurrent combining write for problem sizes $n \geq p^2$. Consider a parallel algorithm for a problem of size $n = \alpha^2 p^2$ ($\alpha \geq 1$) on a p-processor CRCW-ARB PRAM. With each processor simulating α virtual processors in $O(\alpha)$ steps, our algorithm may be used to simulate a concurrent combining write in $O(\alpha p)$ virtual parallel steps, or $O(\alpha^2 p)$ real parallel steps. Any other algorithm for a problem of the same size would require $O(\alpha^2 p)$ steps as well, so that our simulation is optimal when $n \geq p^2$. Using our algorithm, we are able to claim that for p processors:

- A CRCW-PLUS PRAM may be simulated on a CRCW-ARB PRAM with only constant slowdown for problem sizes $n = \Omega(p^2)$.

Our algorithm makes use of two novel techniques: an "overwrite-and-test" memory access method and the creation of a "spinetree" data structure. The "overwrite-and-test" memory access method is an arbitration scheme related to the parallel hashing algorithm of the previous chapter. With this technique, all processors vying for a particular resource send a unique value to a memory location assigned to that resource. If an attempt is made to write many values to the same location, the arbitrary value written identifies the processor that wins the resource. In our multiprefix algorithm this technique is used to build a special tree data structure called a "spinetree" that represents a virtual combining network. By performing these operations only in the first phase of the multiprefix algorithm, all remaining phases proceed with guaranteed EREW memory access.

10.2 Implementing multiprefix

The multiprefix operation is most easily described by a straightforward serial algorithm. After introducing the serial approach, we will explain why it is difficult to parallelize and then introduce our parallel algorithm.

10.2.1 Serial multiprefix

A simple serial algorithm for the multiprefix operation is shown in Algorithm 10.1. The n values are stored in a vector called values with a corresponding label in the vector called labels. The vector multi will hold the multiprefix values computed. The labels are known to lie in the range $[1, 2, ..., m]$. A temporary vector called buckets is allocated to be as large as m to hold the reduction values for each of the labels. Because the labels are integers no greater than m, they directly index sites in the bucket vector. The initialization step clears all needed memory

```
labels :  int[n];
values :  int[n];
buckets :int[m];
multi :   int[n];

SERIAL-INITIALIZATION :
    for (i = 1 to n)
        buckets[label[i]] = 0;


SERIAL-MULTIPREFIX :
    for (i = 1 to n) {
        multi[i] = buckets[label[i]];
        buckets[label[i]] += value[i];
    }
```

Algorithm 10.1: Serial **multiprefix**. A simple serial algorithm for the multiprefix operation. This algorithm exhibits works properly only if the elements are processed in order, making it unsuitable for parallel implementation.

locations, but avoids accessing all of the bucket entries by only clearing those sites referenced by the labels.

The main loop simply processes the elements in order. At each step i, the current value of the bucket referenced by label[i] becomes the multiprefix value multi[i]. Then, the bucket is incremented by the appropriate amount using the increment operator (+=). This loop is similar to the main procedure of a bucket sort, or a general histogramming operation for integer keys, except that those procedures do not save the value of the bucket before incrementing it. These intermediate values of the buckets *are* the multiprefix sums. By extending this operation to arbitrary data types with an arbitrary summing function this algorithm implements a general multiprefix operator.

This loop is difficult to parallelize. It is clear that all elements may not be processed in parallel because elements with the same label would conflict in their modification of the same bucket. Even if accesses to only the same bucket could be serialized a total of n steps would still be required in the worst case when all labels are the same,

10.2.2 Parallel multiprefix

Our parallel algorithm for the multiprefix operation is shown in Algorithms 10.2 and 10.3. Algorithm 10.2 describes the record structure used for temporary values while Algorithm 10.3 describes the four phases of the algorithm. As before, the n values and their labels are stored in vectors value and label. The multiprefix sums are written to multi, and the reductions are left in

```
type spinerec = record {
    rowsum: int;
    spinesum: int;
    multisum: int;
    spine: ptr to spinerec;
};

bucket: spinerec[m];
temp:   spinerec[n];

label:  int[n];
value:  int[n];
multi:  int[n];

INITIALIZE:
    pardo ($i$=1 to $n$) {
        temp[i].rowsum = 0;
        temp[i].spine = &bucket[label[i]];
        bucket[label[i]].spine = &bucket[label[i]];
        bucket[label[i]].rowsum = 0;
    }
```

Algorithm 10.2: The type definition and initialization phase for the parallel multiprefix algorithm. All temporary storage is cleared or set in one parallel step.

the temporary vector called bucket, of size m.

The spinerec record type is used to store the temporary information associated with each bucket and value/label pair in the algorithm. (We will often call a value/label pair an "element" when describing the algorithm.) The spine field is a pointer that connects the elements and buckets into a structure called a "spinetree." The other integer fields will be explained along with the later phases of the algorithm.

In the initialization phase, the temporary storage of each element and the bucket that each element references is cleared as before. The spine pointer of each element is set to the address of its bucket using the address-of operator (&), and the spine pointer of each bucket is set to itself. These operations may be performed in parallel for all elements using concurrent writes and reads of the buckets.

The four main phases of this algorithm will be explained using an example involving 9 elements, all of which have the label 2 and a value 1. With this arrangement, the multiprefix algorithm will enumerate the 9 elements from [0...8] and will place the sum, 9, in the second bucket. The result of the initialization step described is the structure shown in Figure 10.2. All elements direct their pointers to bucket number 2 and the buckets are set to point to themselves.

The 9 elements are shown numbered in vector order but are conceptually arranged into a

```
SPINETREE:
    for ($r = \sqrt{n}$ downto 1)
        pardo ($i = ((r-1)\sqrt{n}+1)$ to $(r\sqrt{n})$) {
            temp[i].spine = bucket[label[i]].spine;
            bucket[label[i]].spine = &temp[i];
        }

ROWSUMS:
    for ($c = 1$ to $\sqrt{n}$)
        pardo ($i = c$ to $n$ by $\sqrt{n}$)
            with temp[i] do
                spine → rowsum += value[i];

SPINESUMS:
    for (r = 1 to $\sqrt{n}$)
        pardo ($i = ((r-1)\sqrt{n}+1)$ to $(r\sqrt{n})$)
            with temp[i] do
                if (rowsum != 0) then
                    spine → spinesum = spinesum + rowsum;

MULTISUMS:
    for (c = 1 to $\sqrt{n}$)
        pardo ($i = c$ to $n$ by $\sqrt{n}$)
            with temp[i] do {
                multi[i] = spine → spinesum;
                spine → spinesum += value[i];
            }
```

Algorithm 10.3: Parallel **multiprefix**. The body of the parallel **multiprefix** algorithm is executed in four main phases. The SPINETREE phase builds the tree of data values. The other three remaining phases then execute with no memory access conflicts.

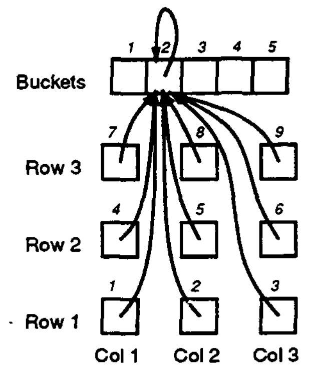


Figure 10.2: The initial pointer structure of the spinetree for 9 elements, each with the label 2. Because only the buckets actually used are initialized, only bucket number 2 is set to point to itself; the other buckets have unknown values in their pointer field.

square. This row and column arrangement is important because later phases will operate in parallel on all of the elements in entire rows or columns in a **pardo** statement. Loops that access rows use a control variable r while loops over the columns use a c. Rows are numbered from bottom to top, and columns from left to right. Given a row r, the elements on that row lie in the range $[((r-1)\sqrt{n})+1,...,r\sqrt{n}]$. Similarly, the elements of column c are given by the sequence $[c, c+\sqrt{n}, c+2\sqrt{n}, ...c+(\sqrt{n}-1)\sqrt{n}]$. These formulas involve simple array address calculations.

This arrangement into rows and columns requires that n is a square. When this is not the case, it is a simple matter to pad the elements up to the next square. A later section will show how this can be avoided in certain circumstances.

The SPINETREE phase links the elements together into the spinetree structure. For each row, from top to bottom, the spine value of each element is replaced with that of its bucket using a concurrent read. Then, by using a concurrent write, the spine pointers of the buckets with elements on the currently active row are overwritten by the address of one of these elements.

This process is illustrated in Figure 10.3 with the initial state of the pointers shown on the left, and the pointer configuration after each row update. The active row is highlighted for each step. After the top row executes this step only the bucket pointer is changed to point to one of the elements of the top row. When the middle row is updated, each element is set to reference an arbitrary element with the same label in the preceding row. Finally, each of the elements of the bottom row readjust their pointers so that they now also reference an element with the same label in the preceding (middle) row.

The pointers now describe a tree with the property that every element is either on a special path called the "spine" or points directly at an element on the spine. In this tree, each child has

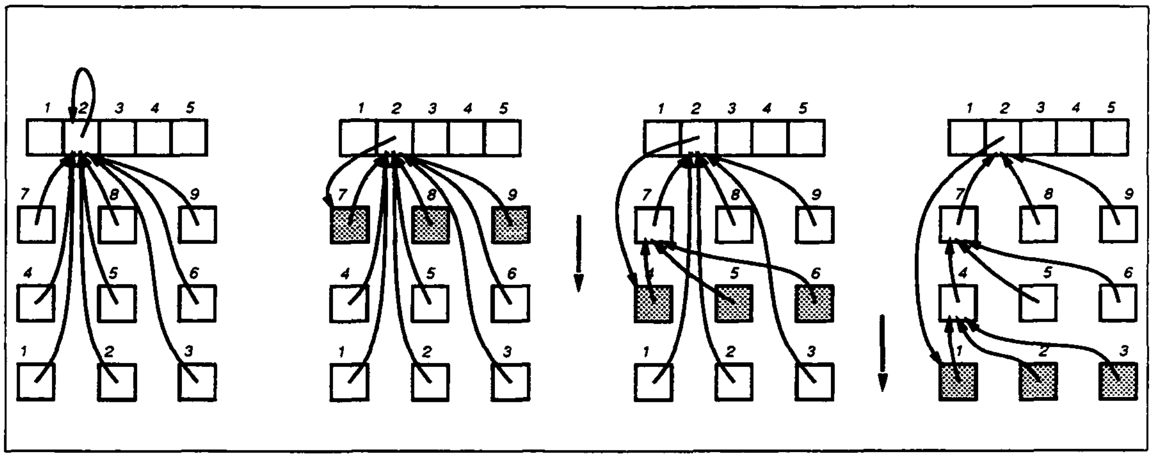


Figure 10.3: The evolution of the spine pointers during the SPINETREE phase of the algorithm. The rows are processed in order with the elements of each first reading the pointer of its bucket, and then attempting to write their own address there. At the end of this process elements with the same label are linked into the spinetree data structure.

a pointer to its parent. A "spine element" is defined as any that has a child in the tree, and the spine is the path that connects the spine elements. In the example, the spine includes elements 4 and 7 and the bucket. (The pointer from the bucket is no longer used and is not considered part of the tree.) Because all elements have the same label (2), they form a connected tree. In general, when there are different labels, each set of elements with the same label (called a "class") forms its own spinetree with its bucket as the root.

The SPINETREE phase uses an "overwrite-and-test" memory access method to determine which elements of a class will be the spine elements. The elements of each row attempt to "overwrite" their bucket to become a spine element. Elements of the next row "test" the bucket pointer value to determine the element of the preceding row that becomes their parent. In this manner, a tree is built such there is only one element of each class on each row that has children. The other three phases use this spinetree data structure to ensure that only one element ever attempts to update its parent in parallel, ensuring EREW memory access.

Snapshots of the intermediate values for each of the elements after the remaining three phases are shown in Figure 10.4. An arrow indicates the order of access by rows or columns. The ROWSUMS phase sweeps across the columns and operates on all elements in a column in parallel. Because the parent of each element must be in a preceding row, elements with the same label in a column will have different parents. By accessing the elements in columns, each element increments the rowsum value of its parent without conflict. At the end of this phase, each spine element is left with the sum of its children in the field rowsum. Elements that are not on the spine will have a 0 rowsum value.

In the SPINESUMS phase, a prefix sum is computed along each spine in the spinesum field.

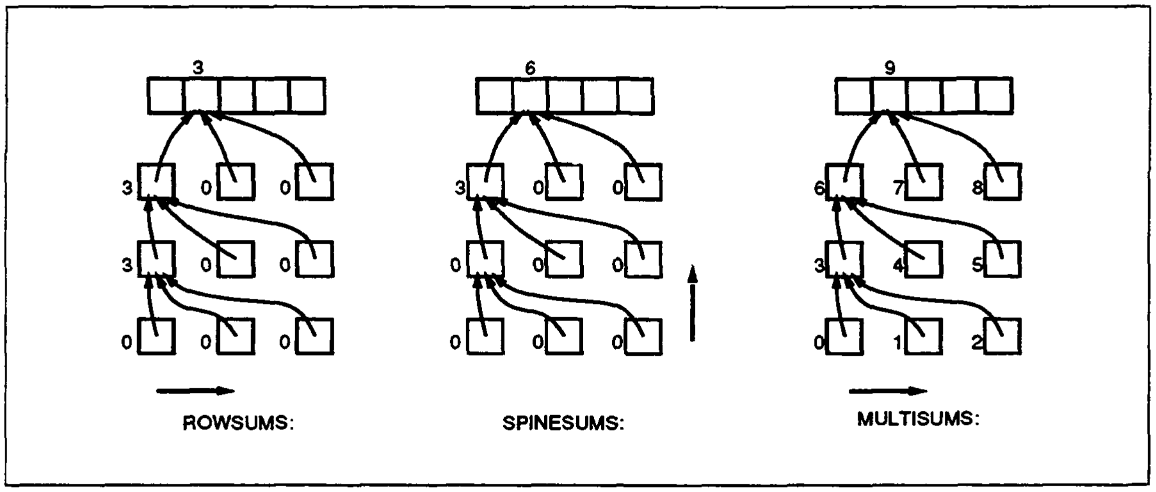


Figure 10.4: Snapshots of the intermediate values computed after each phase. Each of the phases proceeds in a different direction, operating on $\sqrt{n}$ elements in parallel.

Working from the bottom row to the top, each spine element sends the sum of its spinesum and rowsum value to its parent, calculating a recurrence along the spine. Since only spine elements will have a non-zero rowsum value, the conditional ensures that only spine elements participate. Because there is only one spine path through each spinetree, there are no memory access conflicts as children update their parents. At the termination of this phase, each spine element will have in its spinesum field the sum of the elements in its class preceding any of its children. Hence, each bucket will be left with the sum of all elements in its class not including those of the top row. To implement the **multireduce** operation, the reduction value for each class may be calculated directly at this point by adding together the rowsum and spinesum values of the buckets.

In the last phase, called MULTISUMS, the final multiprefix values are distributed to each of the elements. Initially, each spine element has in its spinesum field the sum of all elements in its class preceding any of its children. By accessing the columns in order, each child reads this value as the sum of all elements in its class preceding it, and then increments its parent for the next element of its class on the same row. Once again, access by columns ensures that no two children attempt to update the same parent concurrently. Because the columns are accessed in order, each child is guaranteed of receiving its multiprefix sum in vector order.

The final values shown in Figure 10.4 show the results of a simple multiprefix addition on a vector of 1's with the same label. As expected, the multiprefix operation serves to enumerate these values beginning at 0 and leaves a count of how many values there are in the bucket.

10.3 Algorithmic Analysis

The analysis of this algorithm is straightforward because the memory access patterns are very regular. The initialization phase executes in a single parallel step. Each of the other four phases operates on an entire row or column in parallel with the **pardo** statement in the inner loop. The outer loop iterates over columns or rows for the inner parallel loop. Since there are $\sqrt{n}$ rows and columns, the outer loops of all four phases execute exactly $\sqrt{n}$ parallel steps. The parallel step complexity of the entire algorithm is $S = O(\sqrt{n})$.

The total amount of work performed by this algorithm is the sum over all steps of the number of all elements operated on. The initialization phase obviously performs $W = O(n)$ work. The later phases require $O(\sqrt{n})$ steps and operate on exactly $\sqrt{n}$ elements. Therefore, the work complexity of this algorithm is $W = O(n)$. Because a serial algorithm would also have to perform $O(n)$ work (by visiting all of the elements) this algorithm is work efficient.

10.3.1 Correctness

The algorithm depends on some special properties of the spinetree data structure. This section discusses those properties and shows that they are maintained for any possible labeling of the values. Before continuing, it is important to point out that there is a separate spinetree for each class. Because each spinetree includes only members of its class, a spinetree may skip rows. To further emphasise this point, Figure 10.5 illustrates a possible arrangement of three interleaved trees for the keys 1, 4 and 5. The three trees are completely separate, yet each one of them has the properties of a spinetree. The following section analyzes some of their properties.

Theorem 6 *Elements have the same parent iff they have the same label and are in the same row.*

Proof: For each bucket b, let $R_b = \langle r_k..r_1\rangle$ be the ordered set of rows in which elements have the label b where $r_{i+1} > r_i$. All elements update only their own bucket, so we may consider each class independently. Since the SPINETREE procedure operates in reverse row order, we need only consider for each class b those steps involving rows in R_b in order.

All elements in row r_i with label b replace their spine pointer with that of their bucket and thus have the same parent. Bucket b is then overwritten with a pointer to a member of row r_i. Because the elements may be only a member of one row the pointer values overwritten for each row will be unique and elements will find the same parent only if they were processed together, on the same row.

□

Corollary 1 *The children of a spine element are in different columns.*

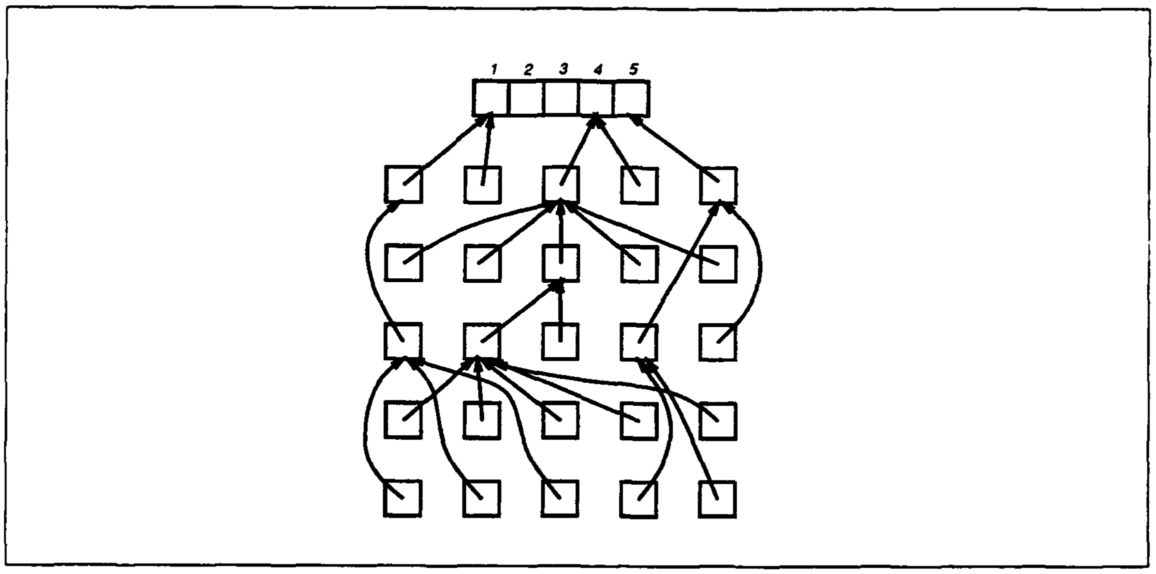


Figure 10.5: Spinetrees may skip rows. A spinetree includes only members of one class. For this reason, its links may not include every row.

Proof: Since the children of a spine element are in the same row, none of them may be in the same column.

□

These properties guarantee that when operating on elements in the same column in parallel, these elements may update their parent with no other member of their class interfering. Also, since each element's parent is in another row, when operating on all elements of the same row in parallel each may modify its parent with the assurance that their parent is not also active. In short, by using the arbitrary concurrent WRITE of the SPINETREE phase a data structure is created in which EREW memory access is guaranteed for the remaining three phases of the algorithm.

Theorem 7 *There is at most one spine element per class per row.*

Proof: A spine element is one that has children. An element is only a candidate for having children if it manages to write its value into its bucket's spine field. This can only be done while its row is active. Because of the write ARB only one element of its class in its row may succeed to become a candidate for having children.

□

Corollary 2 *A spine element has at most one child that is also a spine element.*

Proof: From the preceding theorems. Two children of a spine element could only be spine elements if they were on different rows, and this is impossible since children of the same parent are necessarily on the same row.

□

This corollary ensures that the spine path through the tree for each class has at most one spine element per row. This property is important for the proper functioning of the SPINESUMS phase of the algorithm. In that phase, the spine elements forward their value to their parents. If two spine elements could exist on the same row, or if a spine element could have two children that are spine elements, then there could be write conflicts. Since only spine elements may have a non-zero rowsum value, by processing the rows in order from bottom to top, the sums computed along the spines (the spinesum value) add together only the rowsum values of the spine elements.

10.4 Implementation on the CRAY Y-MP

A vector computer with scatter/gather capability may simulate a synchronous PRAM algorithm by issuing one vector operation for each parallel step [CBZ90]. Using this approach, we implemented a fully vectorized version of our algorithm in C on the CRAY Y-MP. We chose C only because most of our other tools were written in C. This also allowed us to more easily use the C-preprocessor to provide all of the variants of multiprefix we desired with one main template source file. These included such variations as **add**, **max**, **min**, **and**, **or** on data types Integer, Double and Boolean. Full vectorization of the algorithm was achieved by directing the compiler to vectorize each of the inner **pardo** loops. Because the vectorization of these loops is straightforward, it would also be simple to translate the algorithm to FORTRAN.

For our implementation on the CRAY Y-MP we modified the algorithm slightly to use array indexing instead of pointers. This required two simple changes. The first change was to arrange bucket memory to be contiguous with the prefix element memory. This was easy to ensure at the time of allocation. Temporary memory was allocated in one block and divided by a "pivot" point as shown in Figure 10.6. Memory to the left was reserved for buckets, and memory to the right for the elements. In this manner the buckets were addressed by small integers between 1 and m, and the elements were addressed by integers in the range $m + 1$ to $m + n$. Because references to the elements were generally made with respect to a row or column, the conceptual arrangement of the temporary memory is that of a square array with a "handle" for the buckets.

Access of the elements by rows or columns was complicated only by the additional offset. Access by rows, as in the SPINETREE and SPINESUMS phases, simply offset the loop variable i by m. The same change occurred for column access in the ROWSUMS and SPINESUMS phases. By carefully recoding these loops, the compiler was able to deduce that a simple offset was added, as if some references began at an unnamed array allocated at location $m + 1$.

The second change required unpacking the fields of the spinerec record type so that each field was allocated as a separate vector. Instead of one vector of records with four fields, four arrays

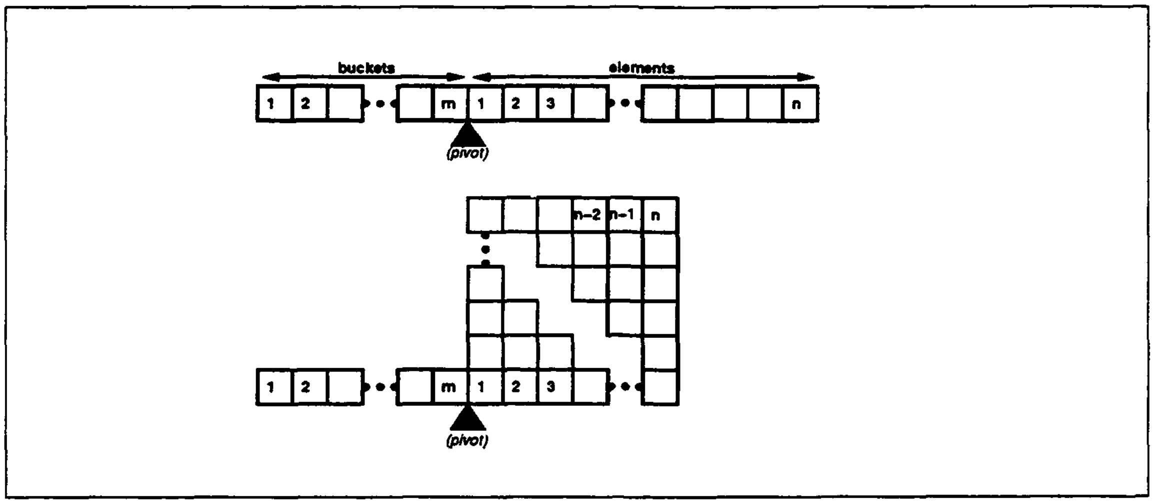


Figure 10.6: Temporary memory for the buckets and elements is allocated in a contiguous block but divided at the "pivot" point. Buckets are located at sites 1 through m, while the elements are offset at positions $m + 1$ to $m + n$. The rows and columns of the elements are indicated by the conceptual arrangement shown.

called spine, rowsums, spinesums and prefixsums were used. With this arrangement, the spinetree data structure could be described by a single vector of length $(n + m)$ of integers no larger than $(n + m)$. Figure 10.7 shows the spinetree structure of the earlier example in its pointer form, and in the integer vector form allowed when the elements are contiguous with the buckets. The vector index of the buckets and elements is shown above each element; it is these indices that the spine values indicate. This allowed the pointer dereferencing operations to be implemented as direct scatter/gather operations using array indexing.

This arrangement also relieved a source of potential memory conflicts that could arise in the record based implementation when referencing the same field in contiguous records. Since the record required 4 words of storage, any sequential access of the same field in the records would result in a memory access with stride 4. Such an access pattern would only make use of $1/4$ of the memory banks available.

One last minor change was made to the initialization phase. In almost all applications the number of buckets, m, is no more than the number of elements, n. In the modified initialization we accessed each bucket directly rather than indirectly through the elements. While this altered the theoretical complexity measure for the real algorithm, in practice it was always faster.

10.4.1 Characterization of the Vectorized Loops

With these modifications, the inner pardo loops could be fully vectorized along the required rows or columns using scatter/gather operations. Because the data dependencies are much too difficult for the compiler to recognize, vectorization directives were inserted to instruct the compiler to

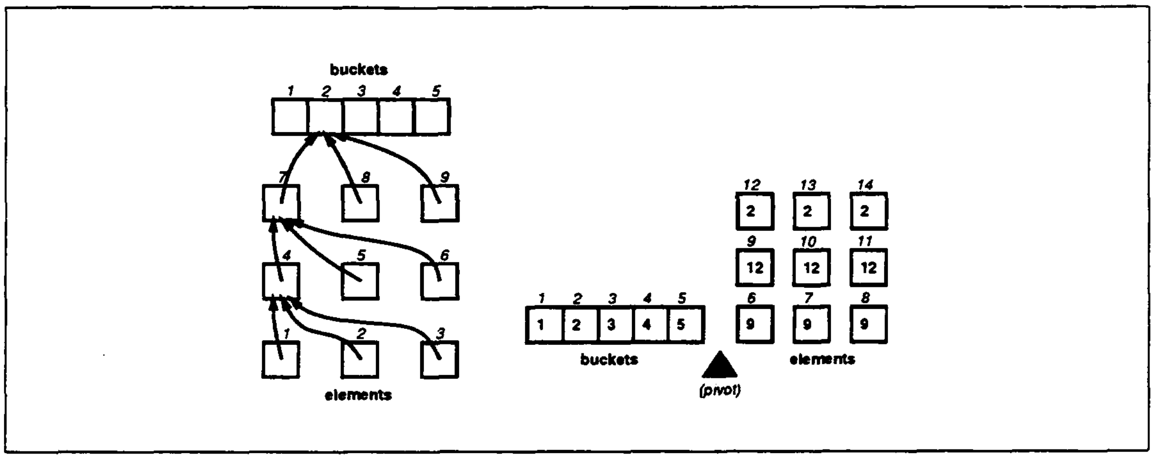


Figure 10.7: A single integer array describes the spinetree data structure. On the left is shown the spinetree from the earlier example. On the right, is its equivalent representation with the index of the elements now offset by m. While the buckets and elements are allocated in one array, it is conceptually divided at the pivot point into the shape shown.

Table 10.1: Vector characterization parameters for the loops of each of the four phases of the multiprefix algorithm. The asymptotic time per element (t_e) is in 6nS clocks per element on the Y-MP. The short vector half-performance lengths indicate that these loops perform well for small problem sizes.

Vector Characteristic Parameters		
Phase	t_e (6nS clk/elt)	$n_{1/2}$
SPINETREE	5.3	20
ROWSUM	4.1	40
SPINESUM	7.4	20
PREFIXSUM	6.9	40

vectorize loops it considered unsafe. Note that while row access is by consecutive elements, column elements are accessed with a constant stride. This section describes the loops involved in each of the four phases and develops performance estimates for them. Of course, since the memory access patterns are data dependent, the figures given represent average cases. However, these have shown to be fairly accurate predictors of performance.

The performance of a vector operation on the CRAY Y-MP may be characterized by the half-performance length ($n_{1/2}$) and the time per element to produce each result (t_e) [HJ88]. With these parameters the approximate time to execute a fully vectorized loop over n elements is

$$t(n) = t_e(n + n_{1/2}).$$

Table 10.1 summarizes the vector parameters for the loops of the four phases as described below. By using the indirect addressing scheme for the spinetree described in the previous section, the coding of each of these loops is straightforward.

1. The SPINETREE Loop

```
for (i = each element of row r) {
    spine[i] = bucket[label[i]];
    bucket[label[i]] = spine[i];
}
```

   The loop shown above is issued for each row of elements. The compiler splits this (using loop fission) into a gather operation followed by a scatter.

2. The ROWSUM Loop

```
for (i = each element of column c) {
    rowsum[spine[i]] = rowsum[spine[i]] + value[i];
}
```

   This loop is executed for each column. The compiler vectorizes this easily, accessing the elements of each column with a constant stride. However, since this loop involves 3 read operations and 1 write and there are only 2 read pipes on the Y-MP, it does not run at peak speed.

3. The SPINESUM Loop

```
for (i = each element of row r) {
    if (rowsum[i] != 0)
        spinesum[spine[i]] = rowsum[i] + spinesum[i];
}
```

   This loop vectorizes but is problematic because of the technique the compiler currently uses for masking. For each group of 64 elements, the compiler first determines which are FALSE. Each of these is given a dummy location to which to send a dummy value. In this way, elements whose rowsum is 0 do not update their parents. Because all dummy values are the same, when there are many FALSE sites the dummy location becomes a hot-spot for memory contention, possibly causing a performance loss. On the other hand, if all 64 elements are FALSE, none of the spine or spinesum values are even read and the loop jumps ahead to the next group of 64 elements. These two opposing effects lead to some strange results.

4. The PREFIXSUM Loop

```
for (i = each element of column c) {
    multi[i] = spinesum[spine[i]];
   spinesum[spine[i]] = spinesum[spine[i]] + value[i];
}
```

   This loop is very similar to the one of the ROWSUM phase but involves one extra write. Because the CRAY Y-MP has only one write-pipe, this operation requires approximately the cost of an additional gather operation beyond the ROWSUM phase.

10.4.2 The multireduce Operation

The **multireduce** operation provides only the reduction values for each label (it is the same as a **send-add** instruction). By modifying the algorithm slightly a multireduce operation is obtained that saves a significant amount of time over the entire multiprefix operation. The key insight is that after the SPINESUMS phase, the reduction values for each label may be calculated directly by summing the rowsum and spinesum value for each bucket.

On the CRAY, this is a simple addition of two vectors and requires only slightly more than 1 clock tick per element. Compared to the PREFIXSUM phase, which requires almost 7 clock ticks per element, this is a substantial savings in time, for only a small modification to the algorithm.

10.4.3 Effects of Label Distribution

The performance of this algorithm is heavily dependent on the density and distribution of the integer labels. The "load" of a bucket is the number of elements in its class. While a heavy average load will degrade the performance of the SPINETREE phase, the same effect may cause the SPINESUM phase to run very quickly. This section describes how these factors interact for different types of data for each of the four phases of the algorithm.

Using a standard pseudo-random number generator to provide labels, we timed the multiprefix operation over a wide range of input sizes and bucket load factors. Our results are summarized in Figure 10.8 with times expressed in 6nS clock ticks per element. Each curve represents a different bucket load factor. A load factor of n indicates that all elements had the same label, while a load factor of 1 means there were as many buckets as elements. However, because of the random number generator used, this does not indicate a one-to-one mapping from elements to buckets.

This data deserves detailed explanation. Three representative cases are explained in below. These are a heavy load (1 bucket), a moderate load, and a light load (n buckets).

Heavy Load: (All labels are the same.) The SPINETREE phase suffers because all scatter/gather operations are to the same memory location, requiring a total of 12 to 13 clock ticks per element. However, after this initially expensive operation, the other three phases run fairly quickly. The ROWSUMS and PREFIXSUMS phases perform as expected, but the SPINESUMS phase exhibits almost superlinear speedup.

Remember that the compiler breaks the rows into chunks equal to the vector length (VL), which is 64 on the CRAY Y-MP. Because only one element per row can be a spine element (since all are in the same class), only one group of 64 elements per row performs any real work, the other groups all exit early. For this reason, this loop runs in as little as 2 to 3 clocks ticks per element, offsetting the expensive spinetree creation phase.

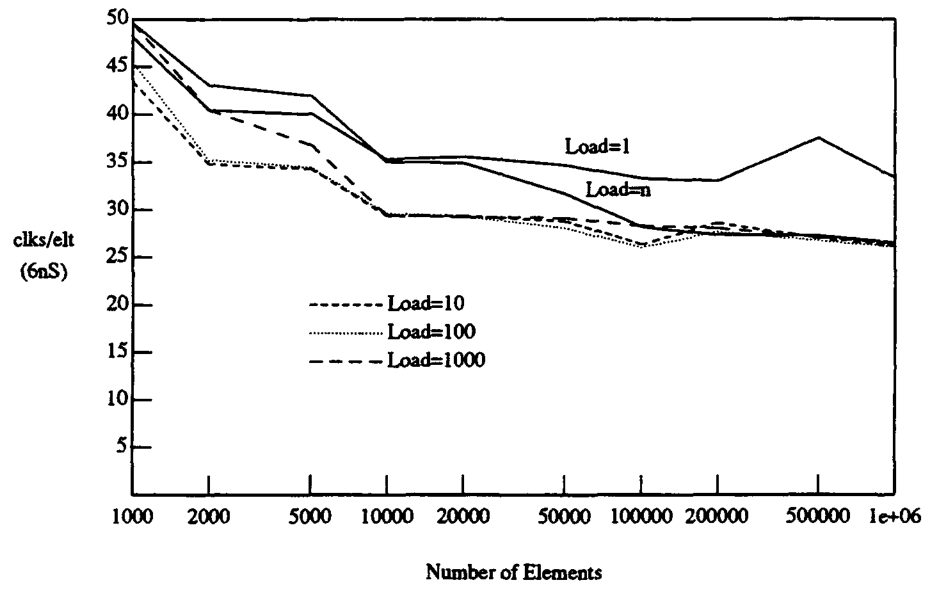


Figure 10.8: The time per element required for input sizes ranging from 1 thousand elements to 1 million. Each curve represents a different average bucket load. A load of n means that only one bucket was used and that all labels were the same. A load of 1 indicates that n labels were randomly distributed over n buckets. Other load factors represent more typical situations.

Moderate Load: This is the region in which the algorithm is most predictable. The performance characterization numbers given earlier for the loops most accurately reflect this situation.

Light Load: Because our CRAY implementation initialized each of the buckets to 0 explicitly, this situation incurs the expense of the additional time required with a very large number of buckets. While the SPINETREE, ROWSUMS and PREFIXSUMS phases all run quite well for this case, it is again the SPINESUMS phase that behaves strangely. Because there are many classes of few elements each in this case, there are few spine elements on each row, but not so few as to gain the superlinear speedup effect. What occurs here is that each group of 64 elements has many FALSE sites which result in the writing of a dummy value to a dummy location. This one location receives values for ALL of the FALSE sites. Because of this memory contention, the SPINESUMS phase runs quite slowly, requiring 8 to 9 clock ticks per element.

What is most interesting about these observations is not the fact that performance varies with label distribution, but that the absolute performance of this algorithm shows little sensitivity to these variations. Even at the far extremes of heavy and light loading, the adverse affects to one phase are offset by the benefits to another. Over input sizes ranging many orders of magnitude, the time per element required varies no more than a few clocks. This fact should assure anyone using this algorithm that it will offer comparable performance for many different applications.

10.4.4 Choosing the row length

In the theoretical PRAM model of the preceding sections, the number of elements n was assumed to be a square. However, the length of the rows and columns may in fact be chosen separately so that their product is slightly greater than n. On the CRAY, the total time of the algorithm is more nearly a linear function of n provided that the loops fully vectorize. Because the loops of the four phases are effectively vectorized, each of the four main phases of the algorithm expend almost a constant amount of time per element.

The main body of the multiprefix algorithm executes in four loops alternatively over the rows and columns of the elements. Given a variable p as the chosen row length, the number of columns will be approximately n/p. The first phase, called SPINE, will then execute in time

$$t_1(n) = t_e^1(p + n_{1/2}^1)\frac{n}{p}$$

while the second phase, which cycles through the columns, will require

$$t_2(n) = t_e^2(\frac{n}{p} + n_{1/2}^2)n.$$

Ignoring the initialization phase, the time for the entire multiprefix algorithm is the sum over the four phases of the times required by each phase.

$$t_{\mathrm{MP}} = t_e^1(p + n_{1/2}^1)\frac{n}{p} + t_e^2(\frac{n}{p} + n_{1/2}^2)n + t_e^3(p + n_{1/2}^3)\frac{n}{p} + t_e^4(\frac{n}{p} + n_{1/2}^4)n$$

The value of p that minimizes this function is found by differentiating and setting the derivative equal to 0. This shows that the total time is minimized when

$$p = \sqrt{n}\sqrt{\frac{t_e^1 n_{1/2}^1 + t_e^3 n_{1/2}^3}{t_e^2 n_{1/2}^2 + t_e^4 n_{1/2}^4}}.$$

For the values of the loop parameters reported earlier, this gives

$$p = 0.749\sqrt{n}$$

indicating that a slightly shorter row length is preferred. (The reason this skewing is so slight is that the loop parameters are fairly evenly matched for the four phases.) However, the sensitivity of this formula to variations in p near the optimal value is very small. For example, using the average case loop performance figures from before, the percent difference between the total time with this optimal row length and a row length of $\sqrt{n}$ is less than 2% when $n = 1000$. For larger n, the total time is even less sensitive.

Because the row length p is the stride used for column access, a more important consideration is the choice of a value that minimizes memory bank conflicts. Our implementation chooses a value near the square root that is not a multiple of the number of memory banks nor of the bank cycle time (4 in the case of the CRAY Y-MP).

10.5 Integer Sorting using Multiprefix on the CRAY Y-MP

The integer sorting problem requires sorting n integers keys whose values lie between 1 and m, for some known m. An algorithm for integer sorting using multiprefix was first described by Ranade in [RBJ88]. The algorithm computes a rank value for each key that gives its position in the final sorted order. Equivalently, the rank of each key indicates how many keys should precede it in sorted order. The entire algorithm is presented in Algorithm 10.4. Because the multiprefix operator guarantees that prefix sums are calculated in vector order, this sorting (ranking) algorithm is stable.

Using the integer keys, the first application of **multiprefix-add** to a vector of 1's provides a count of the number of preceding equal keys for each integer. This step also leaves a count of the total number of each key in the buckets. For each bucket, its cumulative value gives the total number of lower-valued keys preceding it. This vector is calculated with another multiprefix-add operation with all keys being equal. By adding the prefix sums to these values, the final sorted ranking for each key is calculated.

```
key:        int[n];
bucket:     int[m];
cumulative: int[m];
rank:       int[n];

INITIALIZE:
    pardo (i = 1 to $m$)
        bucket[i] = 0;

INTEGER-SORT:
    multiprefix-add(rank, bucket, 1, key);
    multiprefix-add(cumulative, total, bucket, 1);
    pardo (i = 1 to $n$)
        rank[i] = rank[i] + cumulative[key[i]] + 1;
```

Algorithm 10.4: A fast integer sorting algorithm using multiprefix.

Table 10.2: Comparison of Integer Sorting Algorithms on the CRAY Y-MP for the NAS Integer Sorting benchmark. This test involves sorting 8 million 19-bit integers 10 times.

NAS Integer Sorting Benchmark	
Method	Time (Secs)
Partially Vectorized FORTRAN Bucket Sort	18.24
Cray Research Inc. Implementation	14.00
Our Multiprefix-based Sort	13.66

Based on the previous complexity measures, this sorting algorithm has $S = O(\sqrt{n} + \sqrt{m})$ parallel step complexity, and performs $W = O(n + m)$ work. The serial counterpart to this algorithm is called "counting sort" and performs just as much work [Knu68, CLR89], so our algorithm is work efficient.

The NAS Integer Sorting Benchmark

The NAS parallel benchmark suite is a collection of 8 test problems intended to be used to compare parallel machines [BBCS91]. The "Integer Sorting" benchmark requires the sorting of 8 million 19-bit integers. Table 10.2 shows a comparison of the performance times reported for three different approaches on the CRAY Y-MP at the time of writing of [BBB+91]. While the time reported for our multiprefix approach represents a very young implementation with little optimization, it still outperformed the other two approaches.

When timing the NAS integer sorting benchmark on the CRAY Y-MP we took advantage of the fact that each of these applications of multiprefix are simplified cases. In the first call to MP, the values summed are all 1. By using the fact that each value[i] was a constant, the compiler was able to generate more efficient code. This avoided a memory access in each of the ROWSUM and PREFIXSUM loops, allowing them to run faster.

The second **multiprefix-add** is used to compute partial sums across the buckets, all of which have the same label. This operation is a simple prefix-sum, or recurrence computation of the form:

$$c_i = b_{i-1} + c_{i-1}.$$

For the benchmark timing, we resorted to the traditional "partition method" for solving this part of the problem [vdVD89].

Even though both applications of multiprefix in this algorithm are simplified cases, its use is significant. Previous attempts to vectorize the first step of the bucket sorting algorithm have relied on sophisticated compiler technology to recognize this particular loop. We made a simple change to a very general purpose algorithm and achieved excellent performance.

10.6 Summary

This chapter presented a parallel algorithm that implements the multiprefix operation. Our approach introduces some novel techniques. We used the power of the arbitrary concurrent write as an arbitration scheme to build the virtual combining network represented by the SPINETREE data structure. Then, having designed a general purpose synchronous parallel algorithm, we ported it to the CRAY Y-MP by simulating each parallel step with a single vector operation. Since the parallel algorithm is work efficient, it assures that a fully vectorized simulation of the algorithm runs in linear time.

The next chapter presents performance results for **match** and **move** operations built using parallel hashing and the **multiprefix** operator on the CRAY Y-MP. The algorithms considered are numerical sparse matrix kernels. This unconventional symbolic approach to computing on that machine can provide high performance for some important problems.

Chapter 11

Performance trials on the Y-MP and CM-2

This chapter discusses the performance that algorithms written using **match** and **move** can obtain on parallel and vector machines. Our target architectures for these comparisons are the CRAY Y-MP and the Connection Machine CM-2. Of the algorithms presented in Part II, we chose to examine the sparse matrix kernels matrix-vector multiplication, **mvmult**, and matrix-matrix multiplication, **mmmult**. These algorithms were selected because they are typical of important problems run on high-performance computers and many are familiar with them. Because there are aggressive implementations of these procedures there were readily available figures to compare our results to.

11.1 Implementation on the CRAY Y-MP

While the CRAY Y-MP supplies many regular vector operations, it is only with the addition of segmented scans that the full parallel vector model is supported [CBZ90]. To provide the scan operations, we used the CVL (C Vector Library) developed by Guy Blelloch at Carnegie Mellon University [BCSZ91]. In porting the sort-based **match** implementation we used the very fast radix sorting procedure developed by Marco Zagha and Guy Blelloch [ZB90]. On one processor, this integer sorting algorithm is typically five times faster than the standard ORDERS routine of the CRAY Scientific Library [Cra88].

With the CVL library, the CRAY Y-MP provides the CRCW-ARB scan vector model. Concurrent read and write operations are supported directly by the SCATTER/GATHER hardware, but concurrent writes to the same location result in an arbitrary value remaining. Thus, combining-**send** operations are not directly supported. With the addition of the **multiprefix** operation described in Chapter 10, the full CRCW-MULTIPREFIX scan vector model is supported. With these capabilities, the full suite of **match** and **move** operations were implemented using both sorting and hashing-based techniques. Thus, the discussion here includes comparisons between both implementation strategies of **match** and **move**.

The SPARSKIT library is a set of test programs for manipulating sparse matrices. Its implementations of basic sparse matrix operations represent the standard algorithms used by most linear algebra packages. We used SPARSKIT as a baseline from which to evaluate the performance of our approach. The routines we used from SPARSKIT include AMUB (sparse matrix multiply), AMUX (sparse matrix-vector multiplication), and MATRF2, a procedure that generates a random sparse matrix with a given density.

11.1.1 Sparse Matrix-Vector Multiplication

Multiplication of a dense vector by a sparse matrix is at the core of many numerical algorithms. This operation appears when solving systems of linear equations by iterative methods such as the Conjugate Gradient method, and in finite element analysis.

Many elaborate storage schemes have been developed to allow the numerical portion of this algorithm to proceed at near peak speed on vector computers. The Compressed Sparse Row (CSR) storage format is most typically used and arranges the matrix into rows, with the column index of each element stored in a separate vector. This format is very simple and allows the matrix-vector multiply operation to vectorize completely over each row. However, for very sparse matrices, the row lengths can become quite short. Often they are much shorter than the vector half-length of the operation. In an attempt to write loops that better vectorize over more elements, other storage schemes have been developed.

The Jagged Diagonal (JD) format reorders the matrix so that the rows appear in decreasing order of population count. Here, the elements of the re-ordered matrix are collected into groups called "jagged-diagonals." The first jagged-diagonal consists of the first elements of each row; the second, of the second elements, etc. Because the rows are sorted, each of these groups is of successively decreasing length. In this format, each jagged-diagonal begins with an element in row 1, and ends with an element in row k, where k is the length of the jagged-diagonal. Because each of the elements of a group are in different rows, each group may perform a vector update in parallel without the possibility of simultaneous access to the same vector element.

Many sparse matrices consist of just a few jagged-diagonals and the operation vectorizes completely over each jagged-diagonal. The disadvantage of the JD method is its large preprocessing time and the potential problems it has with non-uniform sparse matrices. For matrices with just a few long rows, many of the groups are very short and operations over them vectorize poorly.

The representation used for the **match**-based implementation is known as the full-coordinate format, where every element is stored with its row and column index. Because the **match** approach discovers its own patterns through keys, no particular ordering of elements is required in this format. The implementation of **mvmult** using **match** is able to exploit the fact that the keys used are small integers (allowing hashing to degenerate to a 0-iteration hash procedure), and that there are no **null** values. As explained in the previous chapters, these optimizations save a great deal of time.

Setup Time

The matrix-vector multiply operation can be divided into two parts. The first is the symbolic "setup" time, while the second is the numeric "evaluation" time. Often, when solving systems of linear equations, the same matrix multiplies a vector repeatedly. In such a case, a high setup time can be amortized over many evaluations. It is precisely for this reason that the large setup time associated with the jagged-diagonal format is acceptable for some applications.

The times reported in Table 11.1 are broken down into setup, evaluation and total times. Times measured for the comparisons included only the matrix operation, format conversions were not timed. The order columns gives the number of rows of each matrix and the fraction of non-zero entries is expressed by its density, ρ. The CSR format approach has no setup time associated with it, and performs all of its work during evaluation. The setup time of the JD method is dominated by the sorting of the row lengths and the reordering of the elements into jagged diagonals. In the **match** and **move** approach, the setup time is precisely the time spent building the mappings using **match**. With the sort-based approach, the dominating cost is the sorting step. This is incurred at a fairly substantial cost in time, but allows a fast evaluation step. Because the mappings created involve an index vector, however, the hash-based approach actually performs no iterations. The setup times reported for that approach represent the time spent building the "spinetree" for the **multireduce** instruction of the **move**.

The results reported show that for very large sparse matrices the JD approach trades a large setup time for a quick evaluation. The hash-based approach performs less of its total work during setup, while the CSR approach suffers from very short row lengths for extremely sparse matrices. Because of the speed of the evaluation phase of the JD approach, its use would be preferable in an application that requires repeated multiplication of the same matrix, while the hash-based approach would be better suited to cases where only one multiplication is performed.

The times of the sort-based approach are somewhat disappointing, but understandable. This approach requires much more data movement than the others. A **move** is implemented by sending all of the data values into the agent vector where combining is performed. The result is then permuted again. Most of the other approaches are able to make use of the matrix data in place and do not perform nearly so much data movement. The cost of these permutations is substantial.

While the JD method has the lowest evaluation time, it also has problems with non-uniform matrices. A different series of trials was timed using matrices from electrical circuit simulation problems distributed with the SPARSE sparse matrix package [KSV]. These matrices are very sparse, with an average of only 7 or 8 elements per row, but have a few very long rows. These rows represent power and ground and are almost completely populated. The results are presented in Table 11.2.

In these cases the hash-based approach clearly outperforms both the CSR and JD approaches. This is a known weakness of the JD format, and it has been suggested that such long rows should be handled as a special case [AY89]. In general, the performance of the hash-based approach is more consistent over matrices of varying structure.

11.1.2 Sparse Matrix-Matrix Multiplication

Sparse matrix-matrix multiplication is a lesser used operation in sparse matrix applications. When solving sparse systems, the currently preferred methods are iterative. It is for this reason that there has been so much work revolving around sparse matrix storage schemes that speed up sparse matrix-vector multiplication.

However, as shown earlier, sparse matrix-matrix multiplication is important for the solution of sparse systems using direct methods. Because of the lack of interest in matrix-matrix multiplication, the standard algorithms used are not very efficient. The SPARSKIT library provides one matrix-matrix multiply routine designed for matrices in the CSR format. Because it has a conditional expression in the inner loop, it is not vectorized. This algorithm is typical of the algorithms used for sparse matrix-matrix multiplication.

Once again we used randomly generated matrices of varying sizes. For these trials, the matrix was multiplied by itself. Because the elements of the rows and columns of the matrix are distributed independently and each of the implementations must relate all rows of one matrix to the columns of the other, the use of the same matrix for each operand represents a fair test. This also simplified comparisons. Results of the matrix-multiplication trials are shown in Table 11.3. The number of partial products generated is shown and gives a measure of the total number of multiplications performed.

The most surprising thing about the figures given are how similar they are for the three different approaches, and for the two setup times. The PHASE1 setup time involves matching from the rows of one matrix to the columns of the other and the symbolic portion of the **move-join** operation. Through most of the trials, the times for the hash-based and sort-based approaches are very similar. There is a trade-off occurring here. While the hash-based approach performs little work in the **match** because the keys are simply small integers, the **move-join** operation is slowed by the execution of the **multiprefix** operation. The sort-based approach, on the other hand, spends more time in **match** performing the sorting, but the symbolic portion of **move-join** involves only segmented-**scan** operations, which are faster than the **multiprefix** operations of the hash-based

Table 11.1: A breakdown of the setup and evaluation times of three approaches to sparse matrix vector multiplication for matrices of varying size and density on the CRAY Y-MP. For each category, the best times are highlighted. The CSR approach does no preprocessing, while the JD approach trades a large preprocessing time for a very quick evaluation time. The TOTAL time represents the time required to perform one setup and evaluation. When performing only one matrix vector multiply, the hash-based approach excels for very sparse matrices. Surprisingly, for the largest matrix, the rows are long enough that the CSR approach has the lowest total time.

Sparse Matrix Vector Multiplication (times in mS)													
Order	ρ	Setup				Evaluation				Total			
		CSR	JD	Sort	Hash	CSR	JD	Sort	Hash	CSR	JD	Sort	Hash
20000	.001		37.23	112.87	**12.09**	42.16	**6.61**	27.80	39.08	**42.16**	43.84	140.67	51.17
15000	.001		25.01	69.80	**6.03**	30.11	**3.84**	20.31	21.42	30.11	28.84	90.11	**27.46**
10000	.001		14.85	27.16	**2.67**	19.31	**1.77**	7.97	9.58	19.31	16.62	35.13	**12.25**
5000	.001		6.67	7.61	**0.81**	9.37	**0.43**	2.35	2.61	9.37	7.11	9.95	**3.43**
2000	.005		2.97	6.84	**0.94**	3.87	**0.37**	1.63	3.35	3.87	**3.34**	8.47	4.28
1000	.010		1.52	3.14	**0.36**	1.93	**0.19**	0.91	1.14	1.93	1.71	4.05	**1.49**
100	.400		0.32	1.60	**0.20**	0.27	**0.11**	0.42	0.54	**0.27**	0.44	2.02	0.74
50	1.000		**0.19**	1.00	0.24	0.14	**0.13**	0.32	0.27	**0.14**	0.33	1.32	0.51

Table 11.2: A comparison of the setup and evaluation times for some matrices representing electrical circuits on the CRAY Y-MP. For these matrices with a few very full rows, the JD approach suffers a severe performance loss. The **multireduce** instruction used with the hash-based implementation is better suited to this situation.

Sparse Matrix Vector Multiplication (times in mS)														
Title:	Order	ρ	Setup				Evaluation				Total			
			CSR	JD	Sort	Hash	CSR	JD	Sort	Hash	CSR	JD	Sort	Hash
ADVICE2806	2806	.0030		6.08	8.06	**1.85**	3.60	2.41	3.60	**2.28**	7.99	8.49	11.65	**4.13**
ADVICE3776	3776	.0019		8.13	8.74	**2.10**	7.19	3.21	**2.54**	2.72	7.19	11.34	11.28	**4.82**

Table 11.3: Execution times collected for the sparse matrix multiplication application expressed using **match** versus a traditional FORTRAN code from SPARSKIT on the CRAY Y-MP. The size and density are a measure of the complexity of the **match** from Phase 1 of the algorithm, while the total number of partial products measures the complexity of the final **collapse**. The **match** implementation performed comparably for all trials. This is impressive considering the much higher level of abstraction presented by the **match** operations.

Sparse Matrix Matrix Multiplication (times in S)									
			Setup1		Setup2		Total		
Size	ρ	♯PP	Sort	Hash	Sort	Hash	SparsKit	Sort	Hash
11000	0.001	1.3M	.209	.194	.447	.406	.953	1.060	.932
10000	0.001	1.0M	.146	.156	.305	.272	.729	.668	.743
5000	0.001	127K	.030	.031	.041	.037	.108	.119	.103
2500	0.005	363K	.051	.118	.062	.113	.294	.276	.274
1000	0.010	103K	.016	.017	.038	.033	.075	.087	.078
100	0.040	169K	.016	.018	.038	.052	.107	.103	.113
50	1.000	125K	.012	.013	.031	.036	.078	.080	.079

approach. The end result is that they are fairly equal.

The PHASE2 setup time is dominated by the **collapse** function. Through most of the trials, these times are also nearly the same. The earlier chapter comparing parallel hashing to sorting used the CRAY ORDERS routine for its comparison. The sort used in the implementation of **match** is the much faster sort described earlier. Because of this, the two implementations of **collapse** are nearly equal. These results provide further evidence that hashing is a viable alternative to sorting for some applications.

It is somewhat disappointing that the fully vectorized implementations of **mmmult** using **match** and **move** do not exceed the performance of the non-vectorized matrix-matrix multiplication of SPARSKIT. However, it is impressive that they obtain similar performance considering the much higher level of abstraction presented by the **match** operations, and the fact that the matrices need not be ordered. This parity is partly due to the fact that the scalar performance of the Y-MP is fairly evenly matched to its vector performance, and that the **match** approach performs extra work. On other machines with very poor scalar performance, the **match** and **move** approach would clearly excel. Also, because of use of vector primitives for the **match**-based implementations, it could be easily extended to use the multiprocessor facility of the CRAY Y-MP. The SPARSKIT algorithm is unsuitable for extension to parallel execution.

The setup times shown for the **match** based implementations represent the symbolic processing performed in the two phases of **mmmult** as shown in Algorithm 7.1. As before, these times could be amortized over multiple instances of multiplying matrices with the same non-zero structure. This important optimization could be employed in the sparse Gaussian elimination algorithm

Table 11.4: Evaluation time only of Sparse matrix-matrix multiplication on the CRAY Y-MP. The **match** based implementations naturally decomposes the problem into symbolic and numeric portions. Repeated use of only the numeric portion allows a great savings in time.

Evaluation Time Matrix Matrix Multiplication (times in S)					
Size	ρ	‖PP	Sparskit	Sort	Hash
11000	0.001	1.3M	.953	.404	.332
10000	0.001	1.0M	.729	.217	.315
5000	0.001	127K	.108	.048	.035
2500	0.005	363K	.294	.043	.163
1000	0.010	103K	.075	.033	.028
100	0.040	169K	.107	.049	.030
50	1.000	125K	.078	.037	.030

of Chapter 7. As explained there, it is typical for that algorithm to be broken into three stages: ANALYZE, FACTOR and SOLVE. When solving many systems with the same non-zero structure, but different values, the ANALYZE phase is only performed once. It is during this phase that the mappings of the necessary **mmmult** procedures could be created, for use during FACTOR.

Table 11.4 shows only the evaluation times of the three approaches for the same matrices as before. There is no breakeven point here, because for a single iteration, all approaches are nearly equal. Because the SPARSKIT algorithm can not perform any symbolic preprocessing, the numeric evaluation portions of the **match** and **move** algorithms are faster by up to a factor of 3. These time savings would be significant in the use described above.

11.2 Implementation on the Connection Machine CM-2

The Connection Machine model CM-2 is an instance of a machine to which the parallel vector model applies directly. Because it provides a combining-**send** operation, it provides the CRCW-PLUS vector model. We implemented a full set of **match** and **move** operations based on sorting, but for the hash-based approach could not provide an efficient **move-join** procedure. This is because of the lack of a **multiprefix** instruction on that machine.

Algorithms examined were sparse matrix-vector and matrix-matrix multiplication, as before. While there are published performance figures for dense matrix operations on the CM-2 [JHM89], ours represent the first detailed results reported for sparse matrix multiplication.

Table 11.5: Sparse matrix-matrix multiplication on the CM-2. The symbolic computations reflected in the "join and "collapse" columns dominate the execution time. The time per partial product in microseconds is given in the last column.

Connection Machine Matrix Multiplication							
		Time (S)					
N	ρ	Total	join/move	collapse/move	PP	ρ^2N^3	$\frac{\mu S}{PP}$
20000	0.0005	9.687	1.866/1.243	5.899/0.671	2M	2M	4.8
15000	0.0005	4.387	0.881/0.376	2.808/0.317	737K	844K	6.0
10000	0.001	4.856	0.931/0.618	2.975/0.328	1M	1M	4.8
5000	0.001	0.788	0.230/0.071	0.434/0.052	127K	125K	6.2
2000	0.005	1.230	0.278/0.133	0.739/0.079	203K	200K	6.1
1000	0.01	0.658	0.149/0.062	0.406/0.040	103K	100K	6.4
500	0.02	0.382	0.087/0.046	0.230/0.019	53K	50K	7.2
100	0.4	1.069	0.131/0.191	0.661/0.085	169K	160K	6.3
50	1.0	0.709	0.103/0.099	0.405/0.101	125K	125K	5.7
100	1.0	4.357	0.318/0.491	2.933/0.612	1M	1M	4.4

11.2.1 Sparse Matrix-Matrix Multiplication

The results collected from the matrix-multiplication application are shown in Table 11.5. All trials were run on a 16K processor Connection Machine with test matrices generated by the same procedure as before. Both phases of the algorithm used a sort-based mapping. The total times of the application are much higher than those for the CRAY implementation because of the great expense of communication operations on the CM-2. While our performance figures are not outstanding (especially compared to a dense matrix algorithm), the sparse matrix approach still has merit.

As illustrated in the last column, the time of our algorithm is roughly proportional to the number of partial products generated. With a randomly generated sparse matrix, the expected total number of partial products equals ρ^2N^3. This estimation is very close to the actual number of partial products generated in our trials. By estimating the performance of our algorithm as a function of the number of partial products, the following equation predicts the total time.

$$t_{\text{sparse}} = K_{\text{sparse}}\rho^2N^3$$

Because the dense matrix algorithm must compute all N^3 partial products and sum them, its total time is proportional to N^3.

$$t_{\text{dense}} = K_{\text{dense}}N^3$$

In a comparison with the results reported in [JHM89] for trials run on a 16K Connection Machine, we have found the ratio of the constants to be $\frac{K_{\text{sparse}}}{K_{\text{dense}}} < 2500$ over a wide range of N. While this ratio may seem quite large, it indicates that for very large matrices, the sparse approach excels when

$$\rho \leq \frac{1}{\sqrt{2500}} \leq 0.02.$$

Many applications involving finite element grids and electrical circuit simulation operate on matrices of much smaller density.

Another important observation can be made from the "join" and "collapse" columns. These times include the time of the symbolic part of the join and collapse operation for PHASE 1 and PHASE 2 of the algorithm. As before, these times would be amortized over multiple invocations of the matrix multiplication algorithm applied to matrices with the same structure.

The "collapse" times dominate the cost of the **mmmult** procedure for all of the trials, independent of the sparsity of the system. As explained earlier, because the **move-join** function requires a sort-based mapping on the CM-2, both phases of the algorithm were implemented using a sort-based **match**. However, the second phase of the algorithm, using **collapse**, can be implemented using hashing by simply inserting the appropriate **pragma** statement. By making a simple change we can explore the effect of using hashing to perform the reduction of the partial products in the second phase of the algorithm.

Table 11.6 compares the times between the sort and hash-based approaches. The double hashing strategy was selected because it usually completes in fewer iterations, as shown in Chapter 9. The number of partial products is shown in column "PP" and the number of non-zero entries of the result is shown in column "Size." The ratio between the two gives the average number of partial products summed to form each result.

In all cases but the dense matrix, the hash-based approach exhibited a significant time savings. This savings amounts to 15-50%. This difference is entirely due to the faster **collapse** provided by the hash-based approach because the **move** times associated with each trial are nearly equal. These tests not only confirm the practicality of using hashing in sparse numerical applications, they demonstrate how two different approaches to the same algorithm can be examined with little effort by the programmer using **match** and **move**.

11.2.2 Sparse Matrix-Vector Multiplication

Results collected for the matrix-vector multiplication application are presented in Table 11.7. The same matrices were used as before. The total number of non-zero entries of each argument matrix is given in the "NZ" column. We compare the sort-based **match** implementation to a hash-based version. Because the mappings involve index vectors, the mappings simply describe concurrent **get** and combining-**send** operations, which are directly supported by the architecture. This simple optimization saves a great deal of time in all cases, and is transparent to a program using **match** and **move**.

Table 11.6: Sort versus Hash-based **collapse** for sparse matrix multiplication on the CM-2. The partial products must be generated using a sort-based **match** and **move-join**. Summing of the partial products is accomplished using **collapse** and **move-add**. For all but the dense matrix example, the hash-based implementation of **collapse** is faster.

Sort versus Hash-based Matrix Multiplication							
				Total (S)		collapse/move (S)	
N	ρ	PP	Size	Sort	Hash	Sort	Hash
20000	0.0005	2002970	520541	9.687	8.184	5.899/0.671	4.145/0.911
15000	0.0005	737310	255505	4.387	3.208	2.808/0.317	1.616/0.331
10000	0.001	1002970	260541	4.856	3.929	2.975/0.328	1.933/0.405
5000	0.001	126870	55507	0.788	0.538	0.434/0.052	0.167/0.054
2000	0.005	202970	52541	1.230	0.827	0.739/0.079	0.327/0.086
1000	0.01	102970	26541	0.658	0.493	0.406/0.040	0.242/0.045
500	0.02	52970	13541	0.382	0.180	0.230/0.019	0.029/0.019
100	0.4	169570	9173	1.069	0.633	0.661/0.085	0.233/0.084
50	1.0	125000	2500	0.709	1.545	0.405/0.101	1.245/0.099

There is an interesting tradeoff between the Setup times of the two approaches. If the same matrix multiplies a dense vector repeatedly then the Setup time may be amortized over many applications of the Evaluation portion. The last column titled "Break-Even" shows how many iterations are required before the hashing approach spends more total time than the sort approach. This situation occurs because the combining-**send** operations used with the hash-based mapping are slower than the segmented-**scan** operations used with the sort-based mapping. Depending on the context of the use of the matrix-vector multiply routine, a programmer might select the hash or sort based strategies with a **pragma** statement. This method of choosing an approach allows the exploration of different implementation strategies with very little effort.

Table 11.7: Sparse matrix-vector multiplication on the CM-2. The hash-based approach makes an enormous time savings in two ways. First, no actual hashing steps are necessary because the **match** involves an index vector. Secondly, the **move** uses the combining hardware of the machine directly.

Sparse Matrix Vector Multiplication (times in S)

Order	ρ	NZ	Setup		Evaluation		Total		Break-Even
			Sort	Hash	Sort	Hash	Sort	Hash	
20000	0.0005	200K	0.958	0.002	0.061	0.097	1.019	0.099	27
15000	0.0005	105K	0.587	0.001	0.030	0.044	0.618	0.045	42
10000	0.001	100K	0.490	0.001	0.033	0.049	0.523	0.050	31
5000	0.001	25K	0.167	0.000	0.015	0.021	0.182	0.021	28
2000	0.005	20K	0.179	0.000	0.017	0.023	0.196	0.023	30
1000	0.01	10K	0.101	0.000	0.012	0.014	0.113	0.014	51
500	0.02	5K	0.111	0.000	0.011	0.013	0.122	0.013	56
100	0.4	4K	0.111	0.000	0.013	0.017	0.124	0.017	28
50	1.0	2.5K	0.112	0.000	0.013	0.015	0.125	0.015	56
100	1.0	10K	0.103	0.000	0.016	0.019	0.118	0.019	34

Part IV

Summary

Chapter 12

Summary and Conclusion

This dissertation presents **match** and **move** as architecture-independent parallel primitives of sufficient generality to describe many different types of algorithms. The types of algorithms for which **match** and **move** offer the most leverage, however, are those concerning problems of dynamic structure. This class of problem is especially interesting because they represent some of the most difficult problems for efficient parallel implementation. The difficulties incurred are two-fold. First, the irregularity of sparse structures makes it difficult to map them onto parallel computers, and second, there is a lack of programming support to ease the description of these algorithms.

While most of the algorithms presented in this dissertation have been well documented previously, it is because of the power of the **match** operator that they can be presented here in one unified framework. We presented algorithms on sets, graphs, binary trees and sparse matrices using only a small set of parallel primitives, and were able to show that they obtain the same efficiency measures as more conventional means of implementation. This result was achieved using the much higher level of abstraction maintained by the **match** and **move** primitives.

By illustrating several different implementation strategies for **match** and **move**, we showed that they offer not only architecture-independence, but implementation-independence as well. On any given machine, there may be multiple implementation strategies available. The best suited strategy depends on properties of the algorithm or on characteristics of the data discovered at run-time. The choice of a particular method may be left to a compiler, or to a programmer through the use of hints given by **pragma** statements. This simple mechanism gives the programmer a way to explore different implementation strategies using the same program core.

This extra measure of implementation independence arises because **match** and **move** describe only the desired results of a computation, but not necessarily the method used to compute it. Rather than dictating the policy of implementation, they describe an abstract operation. However, they are not so abstract as to completely hide the underlying machine. The implementation of the operators is sufficiently transparent that a programmer has a good "feel" for the expected performance that a program written using **match** and **move** can offer. In this way, they balance high-level abstraction with a view not too far removed from the actual target hardware.

Lastly, abstraction and high-level operations that ease the programmer's task are fine, but only if they can promise high performance. We demonstrated that algorithms written using **match** and **move** can provide performance rivalling that of more traditional approaches, while remaining high-level and architecture-independent. Of course, a highly aggressive implementation of any algorithm at a low level can outperform an approach that uses general-purpose parallel operators, but we showed that the gap need not be that wide.

12.1 Other Contributions

The main thrust of our research was the efficient implementation of the family of **match** and **move** operations. We used instructions of the scan vector model to construct the algorithms presented, but some of the algorithms are of independent interest. Two main points that are interesting on their own are the parallel hashing and multiprefix algorithms.

Parallel Hashing

Parallel hashing is a technique well-known in the folklore of parallel computing. However, its use seems to be regarded with skepticism because it is a randomized algorithm. We showed that the expected amount of work performed by parallel hashing is linear with the problem size and that with hash-based algorithms good performance can be expected for real problems. These results were supported by both the synthetic trials of Chapter 9 and the results of the sparse matrix-matrix multiplication algorithm of Chapter 11. This work provides convincing evidence that parallel hashing is an important fundamental algorithmic technique for general use in parallel algorithms.

multiprefix

The discovery of the **multiprefix** algorithm was driven by the need for an efficient implementation of **move-join** with a hash-based mapping. However, the algorithm has general utility outside of the implementation of **move-join**. We showed that it may be used to construct a very fast integer sorting algorithm on the CRAY Y-MP. In its **multireduce** incarnation, it also forms the basis of a sparse matrix-vector multiplication algorithm. The **multireduce** instruction has recently been recognized as important enough that the High Performance Fortran Forum is suggesting its adoption as an intrinsic with the name XXX_SCATTER [hpf92]. The **multiprefix** and **multireduce** algorithms could provide an efficient implementation for this intrinsic on other vector machines.

12.2 Future Work

The most immediate question that needs to be answered is how **match** and **move** perform on other architectures. Our work has focused on machines that present a uniform memory model to the programmer. Such a view is central to the scan vector model. Distributed memory architectures or computers with hierarchical memory systems would present a different set of obstacles to efficient implementation.

The vector model has been shown to be applicable to a shared-memory machine. In his dissertation, Chatterjee demonstrated that the data-parallel programming style of the scan-vector model could be compiled efficiently for an Encore Multimax [Cha91]. Because **match** and **move** are operations that build on the vector model, their implementation could make use of the scan-vector implementation of that approach.

However, the best implementation strategy on alternate architectures might possibly dispense with the scan vector model internally. Rather than distributing keys across a vector that is uniformly distributed across the processors, **match** could possibly make use of the locality information present in the specification of groups. The groups described by **match** naturally capture locality information. A better implementation on a distributed-memory machine would migrate the members of a group to the same processor. Because programs written using **match** and **move** do not make use of the ordering of sites in a vector, this operation could be performed transparently.

12.2.1 Non-equi-match

One of the most interesting points that requires examination is the extension of the **match** operator to comparisons other than equality. For example, a **match-lt** (less-than) might relate the keys of the destination to those keys of the source that are lesser valued. The first two questions associated with such an extension are "what does it mean?" and "how can it be implemented efficiently?" Some preliminary ideas are outlined below.

The statement

```
M = match-lt(d.key, s.key)
```

is interpreted to mean that a combining **move** should include values from all sites of the source having keys less than the destination key. A non-equi-**match** implies an ordering of the keys, so that sorting seems a natural implementation strategy. One such approach would append and sort the keys as before. Rather than using a segmented-**scan**, a regular scan would be used to combine the values from the source. In the example of Figure 12.1 the **match** statement links destination sites to *all* source sites with a lesser key. The mapping constructs the same type of agent as before, but it is not segmented. Similar to the regular **move**, a **move-add** executed with this mapping executes two **send** instructions, and one **add-scan**, which is not segmented for

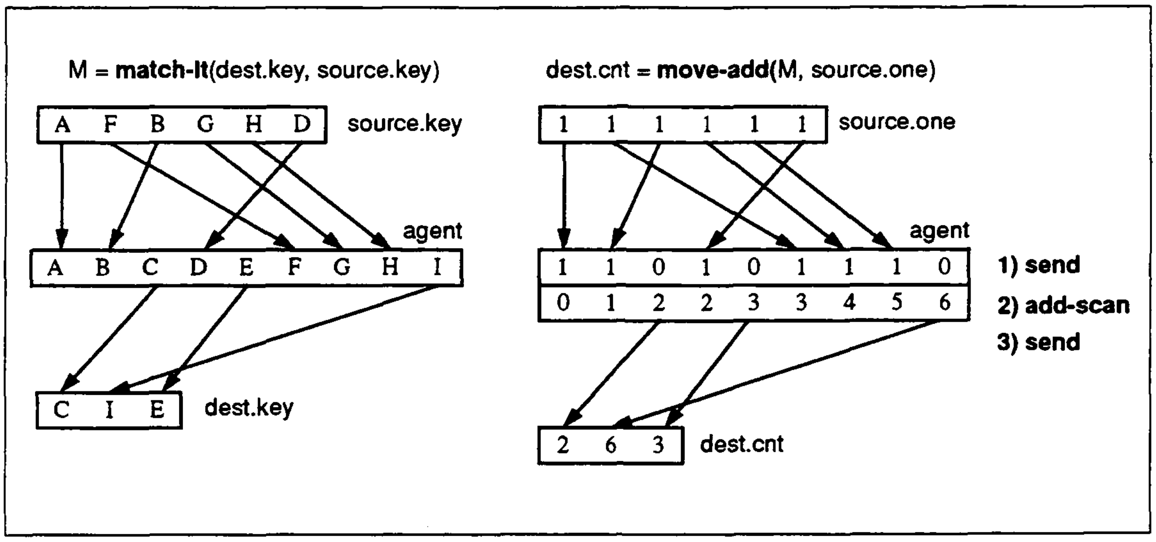


Figure 12.1: A non-equi-**match**. The **match** statement relates destination sites to all those sources with smaller keys. A single **move-add** counts for each destination the number of source sites with a smaller key.

this variant.

The example shown provides each destination with a count of the number of source sites it matched by using a constant vector of ones as the source. The complexity measures for **match-lt** using this approach are the same as that for the regular **match**, and the combining **move** operations would also have the same complexity measures. Extension of this type of mapping to the **move-join** operation seems to be more complicated.

Maximal Point Algorithm

A brief example demonstrates the utility of a non-equi-**match**. The following algorithm for finding maximal points was explained using sorting and scan operations by [MS88]. Using the non-equi-**match** capability it takes just four lines of code.

Definition: Given a finite set S of planar points, a point $p = (p_x, p_y)$ is a *maximal point of S* if $p_x > q_x$ or $p_y > q_y$ for every point $q \neq p$ in S.

Algorithm 12.1 shows a program that finds the maximal points of a set. In our representation, the set of points is stored in a vector set with fields x and y. The mapping, M, relates each point to all other points such that either their X coordinate is greater, or if their X coordinate is equal, then their Y coordinate is greater. In other words, because all of the matching points of each destination have a greater or equal X coordinate, it can only be a maximal point if its Y coordinate is greater than *any* of these that matched. The **move-max** then gathers the maximum Y coordinate of all such points. A point is marked as an extreme point if and only if its Y coordinate is greater than the value computed above.

This short example was presented to show the possible utility of non-equi-**match**. Many such

```
maximal-points(p.<x,y>) {
    M = match-gt(p.<x,y>, p.<x,y>);
    p.L = move-max(M, p.y);
    p.extrem = (p.y > p.L);
    maximal-points = p.extrem;
}
```

Algorithm 12.1: Maximal point algorithm. The use of a non-equi-**match** allows this algorithm to be expressed in only four statements.

geometric algorithms begin by sorting points in the plane to perform similar computations. The approach using a non-equi-**match** helps to express relationships between entities directly, rather than as a side effect of some other operation such as sorting.

12.3 Conclusion

Programming models shape the way people think about algorithms. Traditional serial programs are built from classical data structures and their algorithms. The basic tools every programmer has at his disposal include methods for sorting, selection, traversing linked-lists and binary trees, etc. Using these tools, most programmers begin algorithm design by decomposing the problem at hand into constituent pieces that are more easily handled.

Aggregate data movement operations are as fundamental to the description of parallel programs as basic data structures and their algorithms are to serial programs. As conventional data structures provide a framework within which programmers may organize their thoughts, data movement operations shape the way programmers think about parallel algorithms. As suggested by Sabot, the **match** and **move** operations provide a clear distinction between the description of a communication pattern through **match**, and the movement of data through **move**. By distinguishing these two as separate concerns, programmers are guided to address each separately, and avoid considering too many details at once.

The small set of **match** and **move** operators provide the intellectual leverage necessary for programmers to clearly decompose problems, and also encourage code reuse. By providing these in highly optimized implementations, programmers are relieved of the task of re-designing fundamental parallel kernels for each new task. As new parallel architectures are developed, the adoption of a small set of standard architecture-independent operators will allow the porting of algorithms to new hardware with a minimum of effort.

Appendix A

Expression Syntax used in the Algorithms

The notation used in this dissertation is loosely based on the "C" programming language [KR88]. Each statement is an expression that may produce a vector or scalar as a result. A rich expression syntax makes it easy to express simple calculations concisely. Most arithmetic and logical operators are familiar, but they will be reviewed here for completeness.

Identifiers

Variable, function, and field names are simple identifiers composed of a sequence of letters, numbers and other characters. The special field name # is reserved to signify the index vector. In the examples of this dissertation, function names are written in bold-face to distinguish them from other identifiers.

Comments

A comment line begins with double slashes (//) and terminates at the end of the line.

Labels

A statement may be labeled by preceding it with an identifier followed by a colon (e.g. LABEL:). Labels are never used as the target of a GO-TO, but are used only as a device to convey the purpose of the sequence of statements following the label. In the examples of this dissertation, labels are written in capital letters to further differentiate them from statements.

Control Structures

The looping structures are the familiar **for**, **while** and **do** statements. A **for** statement is followed by three expressions enclosed in parenthesis:

```
    for (expr1; expr2; expr3)
```

and a code block surrounded by braces. The first expression is evaluatcd before beginning the loop. The second expression is the termination condition, and the third expression is evaluated after each execution of the code block.

The **while** and **do** structures test for termination of a loop before or after execution of the code block, respectively. All loops may be exited by executing a **break** statement, which has the effect of jumping out of the innermost enclosing loop.

Arithmetic and Logical Operators

Arithmetic operations are signified by the familiar operators +, −, ×, /. The main difference from "C" is that multiplication is indicated explicitly with × rather than an asterisk, which is used for closure in some of the mathematics presented. The modulo (remainder) operation is represented with %. The logical operators AND, OR are represented by & and |. Logical negation is signified by the unary prefix operator, !.

Comparison Operators

Strict equality is indicated with ==, and inequality by !=. Comparisons on integers and doubles include less-than and greater-than, (<, >), and the variants that include equality (<=, >=).

Assignment Operators

The simple assignment operator, =, evaluates its right argument and assigns it to the identifier of its left argument. Other assignment operators merge in an arithmetic operation. For example,

```
    A += B;
```

evaluates B and adds it "into" A. This operation has the same meaning as

```
    A = A + B;
```

but it avoids evaluating its left argument twice. The assignment operators are extended to include all of the arithmetic operators above.

The Conditional Expression

The conditional expression is a ternary operator.

```
    expr1 ? expr2 : expr3;
```

The first expression must evaluate to a Boolean value. If it is true, then the result of the conditional expression is the value of expr2, otherwise it is expr3.

The Conditional Assignment Operators

With the addition of the **null** value as meaning the absence of a value, the assignment operators are extended to avoid assignments involving a **null** value. Any assignment operator preceded by a ? (question mark) does not alter is left argument if the value of the right expression is **null**. For example:

```
    A ?+= B;
```

will add B into A only if the value of B is not **null**.

Bibliography

[ADD89] P. R. Amestoy, M. J. Daydé, and I. S. Duff. Use of level 3 BLAS kernels in the solution of full sparse linear equations. In *Proceedings of the International Symposium on High Performance Computing*, pages 19 – 31. North-Holland, 1989.

[AG89] S. G. Akl and G. R. Guenther. Broadcasting with selective reduction (parallel machine model). In *Proceedings of the IFIP 11th World Computer Congress*, pages 515–520. IFIP, North-Holland, 1989.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Co., 1974.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, C(27):509–516, 1978.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall International, Inc., 1989.

[And88] Paul B. Anderson. Parallel hashed key access on the connection machine. In Ronnie Mills, editor, *Second Symposium On The Frontiers of Massively Parallel Computations*, pages 643–645. IEEE, IEEE Computer Society Press, 1988.

[AS85] S. K. Abdali and B. D. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science*, 40:257–274, 1985.

[AY89] E. Anderson and Y.Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.

[BBB$^+$91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinki, R. S. Schreiber, H. D Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks — summary and preliminary results. In *Proceedings Supercomputing '91*, 1991.

[BBCS91] D. Bailey, D. Browning, R. Carter, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[BCSZ91] Guy Blelloch, Siddhartha Chatterjee, Jay Sippelstein, and Marco Zagha. CVL: a C Vector Library. School of Computer Science, Carnegie Mellon University, 1991.

[BE88] L. L. Boyer and P.J. Edwardson. Application of massively parallel machines to molecular dynamics simulation of free clusters. In Ronnie Mills, editor, *Second*

Symposium On The Frontiers of Massively Parallel Computations, pages 643–645. IEEE, IEEE Computer Society Press, 1988.

[Bel92] Gordon Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, 1992.

[Bet86] R. Betancourt. Efficient parallel processing technique for inverting matrices with random sparsity. *IEE Proc.*, 133(B):235–240, 1986.

[Ble87] Guy E. Blelloch. Scans as primitive parallel operations. In *Proceedings International Conference on Parallel Processing*, pages 355–362, August 1987.

[Ble88] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models*. PhD thesis, Massachusetts Institute of Technology, November 1988.

[Ble90] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, 1990.

[Ble92] Guy E. Blelloch. NESL: a nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992.

[BM82] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. The Macmillan Press Ltd., 1982.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, November 1990.

[Cha91] Siddhartha Chatterjee. *Compiling data-parallel programs for efficient execution on shared-memory multiprocessors*. PhD thesis, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, 1991.

[Cho88] Kyeongsoon Cho. *Test pattern generation for combinational and sequential MOS circuits by symbolic fault simulation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1988.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1989.

[CMB90] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Workshop on Computer-Aided Verification*, Rutgers University, June 1990.

[Coh88] Evan Reid Cohn. The beta operation: A parallel primitive. Technical Report STAN-CS-88-1231, Department of Computer Science, Stanford University, 1988.

[Coh90] Evan Reid Cohn. Implementing the multiprefix operation efficiently. *Journal of Parallel and Distributed Computing*, 10:29–34, 1990.

[Cra88] Cray Research Inc., Mendota Heights, Minnesota. *ORDERS(3SCI) Manual Page SR-2081 5.1*, 1988.

[Dah90] E. Denning Dahl. Mapping and compiled communication on the Connection Machine System. Technical report, Thinking Machines Incorporated, 1990.

[DNS81] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10(4):657–675, November 1981.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Sofw.*, 8:302–325, 1983.

[Duf65] R. J. Duffin. Topology of series-parallel networks. *Journal of Math. Anal. and Appl.*, 10:303–318, 1965.

[EGK$^+$85] Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin P. McAuliffe, and Larry Rudolph. Issues related to MIMD shared-memory computers: the NYU UltraComputer approach. In *12th International Symposium on Computer Architecture*, pages 126–135, June 1985.

[ESS81] Stanley C. Eisenstat, Martin H. Schultz, and Andrew H. Sherman. Algorithms and data structures for sparse symmetric gaussian elimination. *SIAM J. Sci. Stat. Comput.*, 2(2):225–237, June 1981.

[Gal68] R. G. Gallagher. *Information Theory and Reliable Communication.* John Wiley and Sons, Inc., 1968.

[Geo73] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[Gil80] John R. Gilbert. *Graph Separator Theorems and Sparse Gaussian Elimination.* PhD thesis, Stanford University, December 1980.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York, NY, 1979.

[GLR81] A. Gottlieb, B.D Lubachevsky, and L. Rudolph. Coordinating large numbers of processors. In *Proc. 1981 International Conference on Parallel Processing*, 1981.

[Gol89] Kenneth J. Goldman. Paralation views: abstractions for efficient scientific computing on the Connection Machine. Technical Report MIT/LCS/TM-398, MIT Laboratory for Computer Science, 1989.

[Gon90] Gaston H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison Wesley, 1990.

[Har69] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[HCS79] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, November 1979.

[Hil85] W. Daniel Hillis. *The Connection Machine.* The MIT Press series in artificial intelligence. MIT Press, Cambridge, Mass., 1985.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms.* IOP Publishing Ltd., 1988.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling FORTRAN D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, 1992.

[hpf92] HPFF meeting notes. Notes taken by Chuck Koelbel and Mary Zosel, Sept. 1992.

[HY87] Xin He and Yaacov Yesha. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation*, 75:15–38, 1987.

[JHM89] S. L. Johnsson, T. Harris, and K. K. Mathur. Matrix multiplication on the Connection Machine. In *Proceedings of Supercomputing '89*, pages 326–332, 1989.

[Kan90] Yasusi Kanada. A vectorization technique of hashing and its application to several sorting algorithms. In *PARBASE-90, International Conference on Databases, Parallel Architectures, and Their Applications*, 1990.

[KC90] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. Technical Report CMU-CS-90-148, Carnegie Mellon University, 1990.

[Knu68] Donald Knuth. *The Art of Computer Programming; Volume 3: Sorting and Searching*. Computer Science and Information Processing. Addison-Wesley, 1968.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[Kra90] Steven G. Kratzer. Massively parallel sparse-matrix computations. Technical Report SRC-TR-90-008, Supercomputing Research Center, 17100 Science Drive, Bowie, MD 20715, 1990.

[KRS89] Clyde P Kruskal, Larry Rudolph, and Marc Snir. Techniques for parallel manipulation of sparse matrices. *Theoretical Computer Science*, 64(2):135–157, 1989.

[KSV] Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli. Sparse 1.3, a sparse linear equation solver. Available from EECS Industrial Liaison Program, University of California, Berkeley, CA 94720.

[LBT87] Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Trans. on Computer-Aided Design*, 6(6):981–991, November 1987.

[LD91] Harry R. Lewis and Larry Denenberg. *Data Structures & Their Algorithms*. Harper Collins Publishers, 1991.

[Leh77] Daniel J. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4:59–76, 1977.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[LMN+] Charles E. Leiserson, Jill P. Mesirov, Lena Nekludova, Stephen M. Omohundro, John Reif, and Washington Taylor. Solving sparse systems of linear equations on the Connection Machine. Thinking Machines working paper.

[LPP91] Fabrizio Luccio, Andrea Pietracaprina, and Geppino Pucci. Analysis of parallel uniform hashing. *Information Processing Letters*, 37:67–69, January 1991.

[LRT79] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM J. Numerical Analysis*, 16(2):346–358, April 1979.

[LT80] Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, August 1980.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

[MS88] Russ Miller and Quentin F. Stout. Portable parallel algorithms for geometric problems. In Ronnie Mills, editor, *Second Symposium On The Frontiers of Massively Parallel Computations*, pages 195–198. IEEE, IEEE Computer Society Press, 1988.

[Nor88] Mark J. Norton. Simulating neural networks using C*. In Ronnie Mills, editor, *Second Symposium On The Frontiers of Massively Parallel Computations*, pages 643–645. IEEE, IEEE Computer Society Press, 1988.

[Pan92] Cherri M. Pancake. What should we expect from parallel language standards. *International Journal of Supercomputer Applications*, 6(1):112–117, 1992.

[Pet57] W. W. Peterson. *IBM J. Research and Development*, 1:130–146, 1957.

[Pis84] Sergio Pissanetsky. *Sparse Matrix Technology*. Academic Press Inc. (London) Ltd., 24-28 Oval Road. London NW1 7DX, 1984.

[PMM92] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. MPP Fortran programming model. Technical report, Cray Research, Inc., January 1992.

[Pot88] J. L. Potter. Data structures for associative supercomputers. In Ronnie Mills, editor, *Second Symposium On The Frontiers of Massively Parallel Computations*, pages 643–645. IEEE, IEEE Computer Society Press, 1988.

[PR85] Victor Pan and John Reif. Efficient parallel solution of linear systems. In *Proceedings 17th ACM Symposium Theory of Computing*, pages 143–152, 1985.

[Ran87] Abhiram G. Ranade. How to emulate shared memory. In *28th Annual Symposium on Foundations of Computer Science*, pages 185–194, October 1987.

[RBJ88] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The Fluent abstract machine. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI, Proceedings of the Fifth MIT Conference*, pages 71–93, 1988.

[Ros72] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R.C. Read, editor, *Graph Theory and Computing*, pages 184–218, 11 Fifth Ave, New York, NY 10003, 1972. Academic Press.

[RT78] Donald J. Rose and Robert Endre Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34(1):176–197, January 1978.

[Sab88a] Gary W. Sabot. *An Architecture-Independent Model for Parallel Programming*. PhD thesis, Harvard University, 1988. TR-06-88.

[Sab88b] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. Series in Artificial Intelligence. MIT Press, 1988.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, 2(4):484–521, 1980.

[SH86] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine Lisp: Fine-grained parallel symbolic processing. Technical Report 86.16, Thinking Machines Corporation, May 1986.

[Sha38] Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Trans. of the AIEE*, 57, 1938.

[SJ81] Carla Savage and Joseph Ja'Ja'. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput.*, 10(4):682–691, November 1981.

[SV82] Yossi Shiloach and Uzi Vishkin. An O(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[Tar81] Robert Endre Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[Thi89] Thinking Machines Corporation, Cambridge, Mass. *CM Fortran Reference Manual*, 1989.

[TPH92] Walter Tichy, Michael Philippsen, and Hatcher. A critique of the programming language C*. *Communications of the ACM*, June 1992.

[Val78] J. A. Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, 1978.

[vdVD89] H. A. van der Vorst and K. Dekker. Vectorization of linear recurrence relations. *SIAM J. on Stat. Scient. Computing*, 10:27–35, 1989.

[VTL79] Jacobo Valdes, Robert Endre Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *11th Annual ACM Symposium on Theory of Computing*, pages 1–12. ACM, May 1979.

[Wyl79] James Christopher Wyllie. *The Complexity of Parallel Computation*. PhD thesis, Cornell University, 1979.

[Yan81] M. Yannakakis. Computing the minimum fill-in is NP-Complete. *SIAM J. Alg. Disc. Meth.*, 2(77–79), 1981.

[ZB90] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing*, 1990.